ISSUES IN PETABYTE DATA INDEXING, RETRIEVAL AND ANALYSIS

THÈSE N^{O} 3562 (2006)

PRÉSENTÉE LE 14 JUILLET 2006

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

Laboratoire de systèmes périphériques

SECTION D'INFORMATIQUE

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Sébastien PONCE

ingénieur diplômée ENS de Télécommunications, Paris, France
et de nationalité française

acceptée sur proposition du jury:

Prof. C. Petitpierre, président du jury
Prof. R. Hersch, directeur de thèse
Prof. A. Amorim, rapporteur
Dr F. Machalo Porto, rapporteur
Dr P. Mato Vila, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Lausanne, EPFL
2006

Abstract

We propose several methods for speeding up the processing of particle physics data on clusters of PCs. We present a new way of indexing and retrieving data in a high dimensional space by making use of two levels of catalogues enabling an efficient data preselection. We propose several scheduling policies for parallelizing data intensive particle physics applications on clusters of PCs. We show that making use of intra-job parallelization, caching data on the cluster node disks and reordering incoming jobs improves drastically the performances of a simple batch oriented scheduling policy. In addition, we propose the concept of delayed scheduling and adaptive delayed scheduling, where the deliberate inclusion of a delay improves the disk cache access rate and enables a better utilisation of the cluster.

We build theoretical models for the different scheduling policies and propose a detailed comparison between the theoretical models and the results of the cluster simulations. We study the improvements obtained by pipelining data I/O operations and data processing operations, both in respect to tertiary storage I/O and to disk I/O. Pipelining improves the performances by approximately 30%.

Using the parallelization framework developed EPFL, we describe a possible implementation of the proposed access policies, within the context of the LHCb experiment at CERN. A first prototype is implemented and the proposed scheduling policies can be easily plugged into it.

Key words

High-energy physics	particle collision analysis	cluster computing
two levels indexing	high dimensionnality	data intensive computing
data distribution	cache-based job splitting	delayed scheduling
adaptive delay scheduling	pipelining	scheduling complexity

Résumé

Nous proposons différentes méthodes pour accélérer le traitement des données issues de la physique des particules sur des fermes de PCs. Nous présentons une nouvelle technique d'indexage et d'extraction des données dans un espace comprenant de nombreuses dimensions. Cette technique permet, par l'utilisation de deux niveaux de catalogues, de préselectionner les données efficacement. Nous proposons différents algorithmes d'ordonnancement pour les applications consommant de grandes quantités de données sur des fermes de PCs. Nous montrons que la parallélisation interne des tâches, l'utilisation de caches locaux sur les disques durs des noeuds de la ferme et la modification de l'ordre d'exécution des tâches améliorent considérablement les performances d'un algorithme classique de système de gestion de fermes de processeurs. En outre, nous proposons le concept ordonnancement différé et d'ordonnancement différé adaptatif où l'ajout délibéré d'un délai améliore le taux d'accès au cache local sur disque dur et permet une meilleure utilisation de la ferme.

Nous construisons des modèles théoriques pour les différents types d'ordonnancement et nous proposons une comparaison détaillée de la théorie avec les résultats issus de simulations de fermes de processeurs. Nous étudions les améliorations obtenues par un "pipelining" des opérations d'entrées sorties et des opérations de traitement des données, aussi bien dans le cas de l'accès aux données stockées sur bandes magnétiques que dans le cas de l'accès des données stockées sur le disque dur local. Le "pipelining" améliore les performances d'environ 30%.

Grâce à l'utilisation des outils de parallélisation développés à l'EPFL, nous décrivons une implémentation des différents algorithmes d'ordonnancement proposés ici dans le contexte de l'expérience LHCb au CERN. Un premier prototype a été réalisé et les différents algorithmes peuvent y être aisément intégrés.

Mots clefs

Physique des hautes énergies	Analyse de collisions de particules
Fermes d'ordinateurs	Indexage à deux niveaux
Dimensions élevées	Traitement de grandes quantités de données
Distribution de données	Parallélisation d'applications utilisant le cache
Ordonnancement différé	Ordonnancement différé adaptatif
Pipelining	Complexité de l'ordonnancement

Acknowledgements

Writing the acknowledgments is a good opportunity to look back at the past five years, from the start of my PhD to my current position as leader of the Castor project. However, I've met so many people during this period that it is probably the most complex task of this thesis to thank them without forgetting anyone.

First and foremost, I want to express my deepest gratitude to Professor Roger D. Hersch. As head of the Peripheral Systems Laboratory (LSP) of the EPFL, he accepted to supervise me as a doctoral student at CERN, in the LHCb experiment. Besides the hours he spent reading and commenting the different papers and drafts, he was my guide and the main stimulation that lead this work to completion.

I would like to be equally grateful to my thesis adviser, Pere Mato Vila. As project leader of the GAUDI framework and eminent member of the LHCb computing group, he was my first contact at CERN and my main guide in the world of High Energy Physics. He supported me on a day to day basis during three years and taught me everything about frameworks and software architecture.

More generally, I would like to thank all the members of both the LHCb and the CASTOR team. The LHCb computing team provided the matter of this work and was an invaluable help and support during the first years of my work. Many thanks to Marco, Joel, Philippe, John, Francoise, Florence, Markus, Eric, Gloria, Witek and especially to all my room mates : Radovan, Marianna, Niko and Stefan.

After three years in LHCb, I had the opportunity to move to the Castor development team. Although I was completely ignorant of mass storage systems and large scale applications, I was very welcomed by the members of the team. I want to really thank Olof, Benjamin, Jean-Damien, Emil, Giuseppe, Matthias, Vitali, Victor, Viktor, Paco and Shaun as well as the members of the CASTOR operation team : Tim, Tony, Vlado, Jan, Hugo, Charles and Miguel.

I do not forget the members of the Peripheral Systems Laboratory in EPFL. Even if I was working far from them, they provided me with a lot of support both

from a technical and a material point of view. Special thanks to Sebastian for his support on DPS and Fabienne for the administrative help.

Last but not least, this thesis would not exist without the constant support of my lovely wife, Laurette. Besides enduring the long evenings of debugging and thesis writing, she was also a very precious help. First of all, her deep expertise of LaTeX saved hours of my time. But more importantly, she managed to find clever ways of solving problems by applying her very good sense of physics to computer science. Merci pour tout !

Contents

Acronyms

CERN	Centre Européen de Recherche Nucléaire
DPS	Dynamic Parallel Schedules
EPFL	Ecole Polytechnique Fédérale de Lausanne
HEP	High Energy Physics
LHC	Large hadron Collider : the new CERN accelerator
CMS	Compact Muon Solenoid, one of the four LHC experiment, generic purpose
Atlas	One of the four LHC experiment, with generic purpose
LHCb	One of the four LHC experiment, focusing on the B meson
Alice	One of the four LHC experiment, focusing on heavy ions collisions
event	Collision between two particle beams leading to the creation of new particles
RAW data	Data coming directly from the detector
DST	Data Summary Tape, build from RAW data after processing and containing high level information such as trajectories or particle masses
AOD	Analysis Object Data, data containing the result of the events analysis, e.g. histograms and statistics

Chapter 1

Context and goals

This work has been performed at CERN [wp05b] (European Laboratory for Particle Physics) in collaboration with the LHCb [wp05d] experiment.

In this chapter, we introduce the context of particle physics experiments, their goals and their needs. We will especially concentrate on the computing needs of the particle physics experiments and their implications on the development of Grid oriented models. We will finally describe the challenges, which we will address in the rest of this thesis.

1.1 LHCb, a CERN experiment of HEP

1.1.1 High Energy Physics (HEP) goals and tools

High energy physicists try to understand the fundamental constituents of matter, what they are and how they interact with each other. Since Einstein and its famous equation $E = m\,c^2$, scientists know that matter and energy are equivalent and that a high concentration of energy can turn into matter under the form of elementary particles. For common levels of energies, common particles are generated. These are well known and studied.

The particles of interest are those that existed in very early ages after the big bang [Sin05]. At that time, a huge amount of energy was available, allowing heavier particles to exist. The particle physicists try to recreate these conditions by colliding particle beams, that are previously brought to high energies using a particle accelerator. The particles created from the collisions are then detected and studied in particle detectors. Particle detectors and accelerators are amongst the

world's largest and most complex scientific instruments. For example, the CERN accelerator is 27 km long and the LHCb detector issues 950000 values per particle collision.

1.1.2 CERN : Centre Européen pour la Recherche Nucléaire

CERN is the European Organisation for Nuclear Research, the world's largest particle physics centre. CERN exists primarily to provide physicists with the necessary tools, i.e. with particle accelerators able to reach the highest energy possible.

CERN is currently building a new particle accelerator called LHC (Large Hadron Collider) that will enable to carry out research deeper into matter than ever before. LHC occupies a 27km long tunnel, located 100m beneath the Swiss-French border, near Geneva. It is from many points of view one of the largest and most complex scientific instruments ever built.

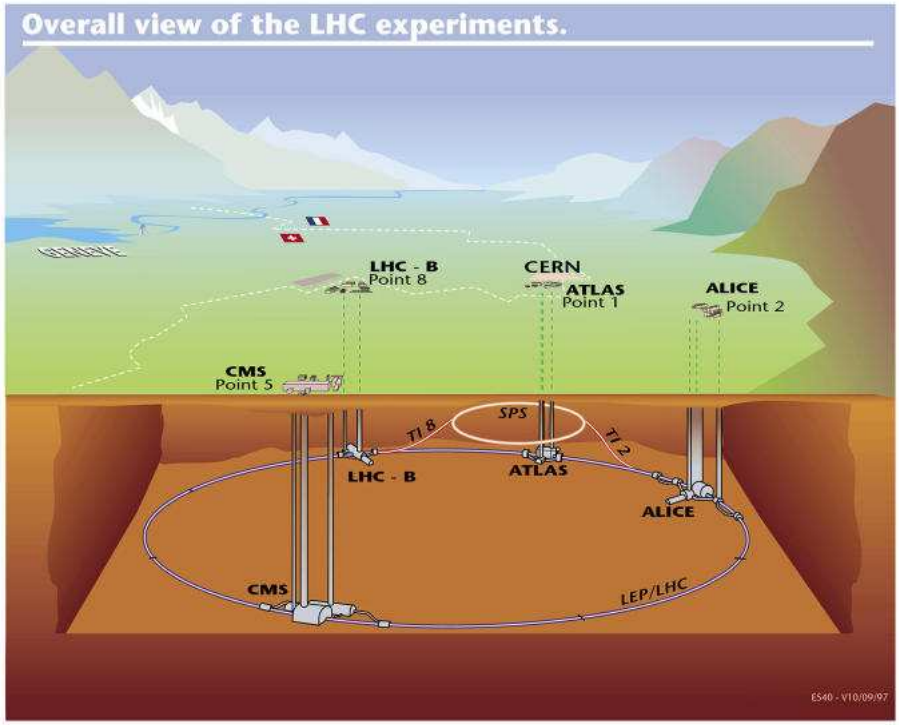

Figure 1.1: The LHC accelerator

LHC will be able to accelerate either two proton beams or two ion beams. The beams will collide every 25 ns, i.e. at a frequency of 40 MHz. Each collision (called event) will create many particles (few thousands). Four main experiments will analyse these particles : Alice, Atlas, CMS and LHCb. Two of them (Atlas and CMS) are general purpose experiments, aiming at studying a wide range of particles. The two others are concentrating on a specific physics question.

- Alice's aim is to study the formation of a new phase of matter at extreme energy densities, the quark-gluon plasma.
- LHCb studies the physics of the B meson and its implication in CP violation [wp05c] (see Section 1.1.3).

Each of the four experiments is building currently a dedicated particle detector.

1.1.3 The LHCb experiment

LHCb [wp05d] is the name of one of the future LHC experiments. Its primary goal is the study of the so called CP Violation [wp05c]. This physical theory suggests that, in the world of subatomic particles, the image of a particle in a mirror does not behave like the particle itself [wp05a]. In other words, matter and antimatter may have a slightly asymmetric behaviour. One of the fundamental reasons of this effect is the existence of the bottom-quark and its cousin the top-quark. It is precisely this bottom-quark, under the form of the B-meson that the LHCb experiment intends to study.

B-mesons are produced by the collisions of the high energy proton beams of the LHC. Each of the forty millions collisions that will take place every second will produce thousands of new particles in which the LHCb physicists will try to detect B-mesons and to measure their parameters and behaviour (especially the way they decay).

The different sub-detectors constituting the LHCb detector are able to detect all created particles and to specify their energy and momentum. Figure 1.2 gives an overview of the detector setup :

- the **vertex detector** provides precise measurements of track coordinates close to the interaction region. See [Col01b].
- the **RICHs** are Ring Imaging Cherenkov devices, able to measure the energy of particles using their cherenkov radiation in a specific medium. See [Col00b].

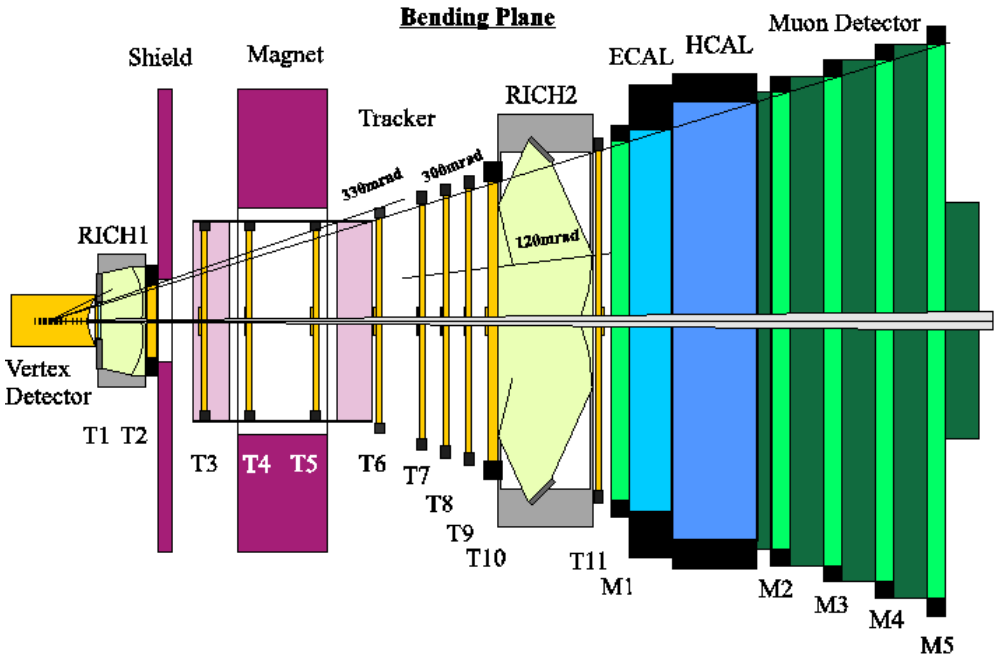

Figure 1.2: The LHCb detector at CERN

- the **trackers** locate particles and provide data for deducing their trajectories. Together with the magnet that bends the trajectories, they also provide a measure of the momentum of particles. See [Col02] and [Col01a].
- the **calorimeters** (Electromagnetic for ECAL and Hadronic for HCAL) stop the particles and measure their energy. See [Col00a].

The global output of the LHCb detector is about 1 MB of data per event at a rate of 40MHz, across 950000 channels. This yields 40 TB of data every second.

Most of this data will not be stored since more than 99,9999% of it is not of interest. Actually, the detector has a four level trigger system that allows a reduction of the data rate from 40 TB/s to 120 MB/s per second (three hundred thousand times less). This factor is obtained by both a reduction of the event size to the order of 600 KB and by a reduction of the event rate to 200 Hz. Assuming that the LHC will run 24 hours a day and 7 days a week, LHCb will produce an order of 10^{10} events per year, which is six petabytes of data (1 PB = 10^{15} bytes).

Table 1.1 summarises the figures concerning the data being saved, indexed and later retrieved by physicists for analysis.

Size of a data item associated to a collision event	600 KB
Nb of collision events	10^9 to 10^{10} per year
Global size of the data	$\sim 6\ 10^{15}$ bytes = 6 PB per year
Data items input rate	200 Hz
RAW data input rate	120 MB/s

Table 1.1: Figures concerning LHCb data

1.2 LHCb data analysis : a GRID world

The data Grid projects [Dat05, EGE05, Eur05] represent the newest developments in the area of distributed computing. This evolution was essentially pushed by the ever increasing computing needs of the simulation and analysis tools developed for different scientific communities such as biologists, meteorologists and particle physicists.

The analysis of the huge amount of data generated by LHCb requires the use of such a distributed, high performance computing grid.

1.2.1 The LHCb data

LHCb data can be categorised in 2 ways : by their type (RAW, DST and AOD) and by their origin (real or simulated data).

Real data come from the particle detector under the form of RAW data. These data items are the direct output from the different parts of the sub-detectors. They contain mostly positions and amounts of energy loss. The particle trajectories, mass and momenta are then reconstructed using dedicated software and stored as DST (Data Summary Tape). These data items are finally analysed by the physicists and the analysis results are stored as AOD (Analysis Object Data) data objects.

Besides real data, the analysis jobs require large amounts of simulated data items (see Section 1.2.3). Simulated RAW data are computed from a simulation of the full particle detector. Reconstruction of the simulated data is carried out using the same tools as for the real data. Creating AOD data objects do not make sense for simulated data.

1.2.2 LHCb Computing strategies

The analysis of the LHCb data will be performed among 47 different physics institutes around the world. The real data generated at CERN will thus be replicated at several places. Simulated data are produced among all the collaborating institutes around the world. Figure 1.3 shows the hierarchy of the institutes as well as the type of data stored in each of them and the expected data transfer rates.

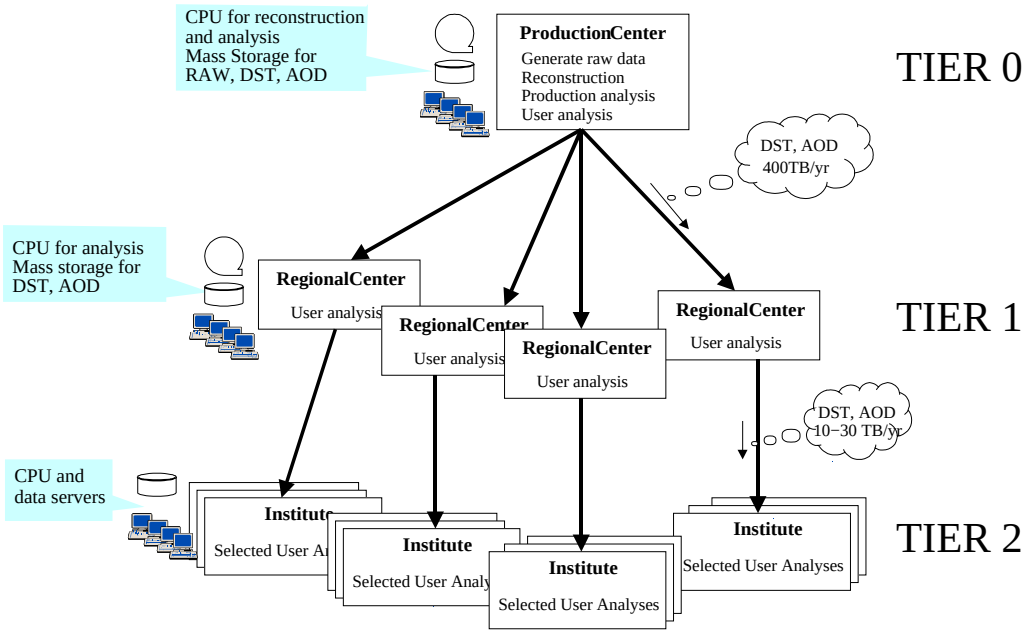

Figure 1.3: The LHCb computing model

The hierarchy has three levels :

- Tier 0, Production centre : this is the only centre producing real data, i.e. CERN. These data are reconstructed and partially analysed in situ. The production centre has to provide mass storage for all the data produced, i.e. RAW, DST and AOD. It also provides processing power (CPUs) for the reconstruction, the analysis and the production of the simulated data.
- Tier 1, Regional centres : these centres held a partial copy of the data, i.e. a full copy of the AOD and a partial copy of the DST. They provide the mass storage systems accordingly as well as the processing power for the analysis. Besides that, they provide processing power and mass storage systems for the production of simulated data.

- Tier 2, Institutes : they mostly provide processing power for analysis and simulation and use the data of their regional centre. If needed, they can copy a selected part of these data items onto their own data servers.

Regional centres and possibly institutes are producers of simulated data. In the case where an institute is a data producer and has no data archiving capability, the corresponding regional centre will store the data and fulfil the role of production centre for the distribution of these data items.

1.2.3 Data analysis over the Grid

Analysis of LHCb data consists in collecting statistics on many physical events and comparing it to simulation results in order to see whether the theory behind the simulated data explains reality.

As a consequence, each analysis job requires the scanning of huge quantities of data items which are usually stored in third level storage systems, on tape or on optical disks, at the level of the production or regional centres. These storage systems usually have disk pools acting as caches and excellent network connectivity but their third level storage systems offer only slow access to the data items.

Analysis jobs are also very CPU intensive and a single analysis job would last months on a single standard computer. Thus, all institutes are equipped with large clusters of computers on which analysis jobs can be parallelized.

All the storage facilities and CPU clusters are made available to physicists through a computing grid. The end user provides to this grid an analysis code and a list of data items to analyse. The analysis job is split and run on any of the computer centres depending on the availability of their respective CPUs and of the corresponding data items.

1.3 Challenges

1.3.1 Indexing

When an analysis job aims at verifying a given aspect of theory, it needs to scan events that deal with the corresponding phenomenon. Selecting the right events is thus essential to reduce the size of the input data. However, the selection has to be done in a high dimensional space over tens of billions of items. Moreover, the selection is an iterative process that needs to be refined several times according to

the result of analysis. It is thus mandatory to be able to select large sets of events in a reasonable time.

In order to achieve this, usual indexing techniques are not sufficient. For this type of usage, standard indexes would be extremely large due the high number of items. Because of the very high dimensionality of the data, a simple sequential scan may outperform best known general purpose algorithms, as shown by Weber et al [WSB98]. We propose a two level indexing scheme to circumvent these problems. It improves the sequential scan by replacing it by a scan of a preselected subset.

1.3.2 Cluster computing

Distribution and partitioning of analysis jobs over the grid is carried out with the help of grid job scheduling software [FK97]. Some tools like the Grid-Brick Event Processing Framework (GEPS) [APF+03] were developed to take into account the data locality in the Grid context so as to optimize the job scheduling. However local parallelization within the cluster of a given institute or within a grid location relies on simple computer farm approaches. Data-intensive parallel processing is different from conventional parallel processing [NEO03] an existing computer farm approaches do generally not take into account the locality of data.

Due to the specificity of the LHCb software, i.e. its large set of input data items, its small set of output data items, its very high parallelization potential and the very low dependencies between subjobs, we propose to study and implement a dedicated parallelization facility that makes use of these specificities and tries to maximise the performances. In particular, we propose to make use of the storage capabilities of each computing node (i.e. the local hard disk) to cache data and to schedule subsequent jobs based on the cache content. We also present scheduling algorithms that improve performance of data retrieval by reshuffling the requests and by scheduling requests at fixed time intervals.

1.3.3 Simulation of event processing on a cluster of computer nodes

The parallelization and scheduling algorithms described in the previous section cannot be tested in a real environment since the computing grid is not yet ready and the institutes don't have their clusters ready either.

A compromise is to simulate the behaviour of the clusters via a dedicated

simulation engine. However, available simulation engines are not adapted to the specificity of the LHCb analysis. They simulate communications between nodes and are therefore too slow to simulate days or weeks of particle collision event processing. For this reason, a dedicated, event based simulation software was developed that does not deal with the details of inter node communications.

1.3.4 Adaptation of existing applications to cluster computing

Most of the reconstruction and analysis software of the LHCb experiment was not developed with cluster computing in mind but as an isolated process running on a single machine. Adapting it to run on computing clusters would thus be a complex task.

However the LHCb applications are based on a very modular and easily extensible framework called GAUDI [cg05], which facilitates this adaptation. The parallelization can be carried out by providing parallel services to the GAUDI framework that replace the original services. In our case, parallelization of any GAUDI based software can be carried out by replacing a single core service called "Event Loop Manager" by its parallel counterpart. This requires rewriting a few hundred lines of code.

We implemented a first parallel version of the "Event Loop Manager" which brings cluster parallelization to any application based on the GAUDI framework. In order to handle the parallelization, the new component uses internally the DPS (Dynamic Parallel schedules) software [GH03] from EPFL.

This parallel implementation of the "Event Loop Manager" can reuse the scheduling algorithms developed in the rest of the thesis within a simulation context. The expected gains and limitations are thus given by the simulation results.

Chapter 2

Indexing

We present here a new way of indexing and retrieving data in huge datasets having a high dimensionality. The proposed method speeds up the selection process by replacing scans of the whole data by scans of matching data. It makes use of two levels of catalogues that allow efficient data preselections. First level catalogues only contain a small subset of the data items selected according to given criteria. The first level catalogues allow to carry out queries and to preselect items. Then, a refined query can be carried out on the preselected data items within the full dataset. A second level catalogue maintains the list of existing first level catalogues and the type and kind of data items they are storing.

We establish a model of our indexing technique and show that it considerably speeds up the access to LHCb experiment event data.

Section 1.1 presents the issues raised by the LHCb experiment in terms of data indexing and retrieval. Section 2.2 presents search results in the domain of data indexing and emphasises their respective strengths and weaknesses. Section 2.3 presents the proposed indexing scheme and shows how it can be used efficiently for data retrieval. Section 2.4 evaluates the performance of the new indexing method compared to sequential scan[1]. Section 2.5 draws the conclusions.

[1]Our method seems to be the only method which, according to our knowledge, overperforms sequential scan for the access of data items in a very high dimension space, according to a large number of criteria

2.1 Issues

The LHCb data types are described in Section 1.1.3 and a summary is available in Table 1.1 on page 19. These data items are analysed off-line by physicists in a rather specific way : the analysis is mainly based on an iterative process consisting in selecting some data items (typically of the order of 10^6) with rather complicated selection criteria, downloading the items, running some computations on them and modifying the selection criteria. A criterion may for example make use of the energy of the event, of the types of the particles involved or of the number of decays. The number of iterations is rather small (in the order of 10) but the selection of the data still appears to be the key of the analysis.

Another issue is the number of indexes that a given criterion uses. It typically uses 10 to 30 parameters with a mixture of numeric, boolean and strings data. These indexes are not always the same for all criteria but a few criterion types can be defined (less than 10) for which the set of parameters is fixed. Due to the high dimensionality of the event data (10 to 30 indexes), up to now, at CERN, the only data selection algorithm was a linear scan of the whole dataset.

2.2 Related Work

There are not many research approaches addressing the issue of indexing data in a high dimension space. Weber et al [WSB98] show that there exists a dimension over which any algorithm is outperformed by a sequential scan. Experiments show that the sequential scan outperforms the best known algorithms such as X-Trees [BKK96] and SR-Trees [KS97] for even a moderate dimensionality (i.e. ~ 10).

The X-Trees and SR-Trees algorithms are based on data partitioning methods. The ancestor of the data partitioning method is the R-tree method [Gut84] which was further developed under the form of R*-Trees [NB90]. However, these data partitioning methods perform poorly as dimensionality increases due to large overlaps in the partitions they define. This is due to an exponential increase of the volume of a partition when the number of dimensions grows.

The SR-Tree [KS97] method tries to overcome this problem by defining a new partitioning scheme, where regions are defined as an intersection of a sphere and a rectangle. The X-Tree [BKK96] method, on the other side tries to introduce a new organisation of the partition tree which uses a split-algorithm minimising overlaps. The results are good at low and moderate dimensions but are outperformed by a

sequential scan for dimensions larger than 10.

However, the variety of useful selection criteria on a given set of data is far from being infinite. They can usually be reduced to a small number of indexes, say 20 to 30 maximum (which is already a very high dimension). Thus, from all values contained in a data item (tens of thousands in some cases), only this very reduced subset of 20 to 30 values is relevant for the selection criteria.

This property is used to define a new indexing method based on two levels of catalogues. This method greatly speeds up the linear selection process by replacing scans of the whole data by scans of matching data. Data is efficiently selected using both server side and client side preselections and the power of the SQL query language.

2.3 A two level indexing scheme

The aim of the proposed scheme is to allow most of the selection to be carried out using catalogues (tag collections) that contain only a part of the data items and, for each item, only a subset of its values (a tag). Several catalogues are built, each one for a different type of query. This enables performing a very efficient preselection of the items before accessing the real data items.

2.3.1 Tags

A tag is a subset of a data item comprising several parameters plus a pointer on this data item. A pointer is simply the information needed to find and retrieve the data item, be it a regular pointer (memory address), a file name, an URL or another information.

A tag contains few values (also called parameters) of the data item that are used as selection criteria. For a given criterion, or even a given type of criterion, the number of tag values is small (10 to 30) which results in a tag size of 10 to 200 bytes. For example, in the case of some physics events, one may want to include in the tag the energy, the nature of the event and the number of particles involved.

Several types of tags can be defined, with different sizes and contents, even for the same data items. Different tags will point to different subsets of the data items and correspond to different criteria.

Tags are small, well structured objects that can be easily stored in a relational database. Thus, they can be searched using the power of SQL-like languages. The storage of tags in a relational database is trivial : each type of tag is stored in a

different table, whose columns are the different values included in the tag plus one for the pointer to the real data item. The data item itself does not need to be part of a database.

Tags will be used to make preselections without loading the data items. Since a tag is approximately 200 bytes and an event 600KB, we reduce the amount of accessed data by a factor of $3\ 10^3$ for the case of the LHCb experiment.

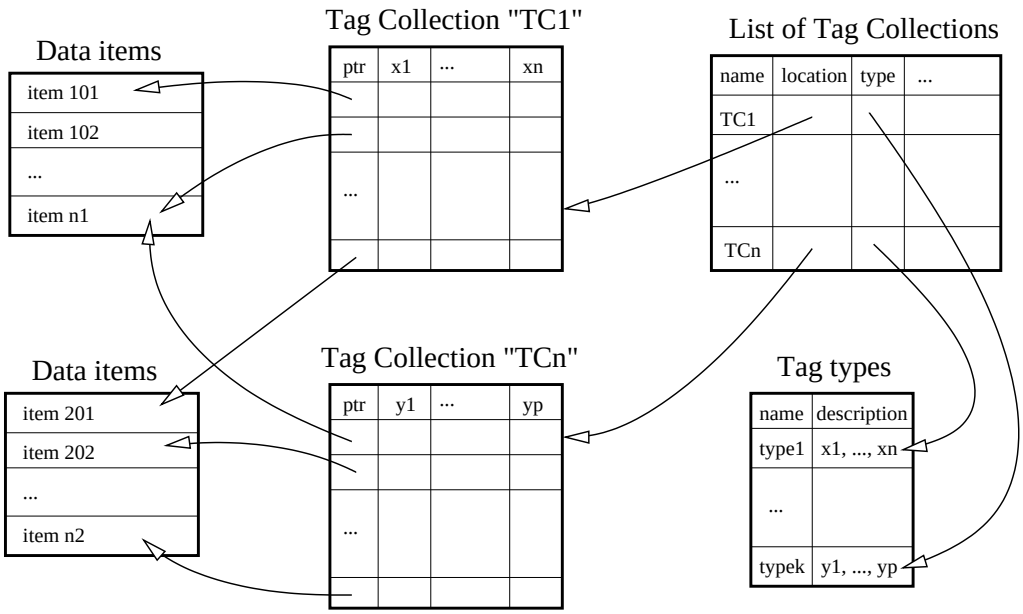

Figure 2.1: Structure of the tag collections

2.3.2 Tag collections

As explained above, tags are subsets of collision event data items. A tag collection is a set of tags, all of the same type. It corresponds to a set of data items but with only a subset of the data items values. The values themselves fulfil some criteria, such as being in the interval between a minimal and a maximal value. Thus, two different tag collections may correspond to two different subsets of data items, even if they use the same set of values (type of tags). These subsets may of course overlap.

Tag collections are stored in a relational database as a table, where each line is a tag and columns correspond to the values contained in the tags (Fig. 2.1). The

different tag collections are assembled into a list of tag collections, each with its associated tag type.

Since tag collections only contain tags for a given subset of the data items, they act as a first preselection of data. For example, in the LHCb experiment, the tag collections are in the order of 10^5 smaller than the database, i.e. approximately 10 GB. A factor 10^3 is due to the tag size (Section 2.3.1) and another 10^2 factor comes from the fact that, on average, less than 1% of the data items have a tag in a given collection, i.e. tags whose values are within the predefined ranges associated to that collection. Thus, a collection has typically 10^7 to 10^8 entries.

Collections can be defined by any user or group of users who wants to be able to use a new selection criterion. The creation of a new collection may either require a scan of the full set of data items or is extracted as a subset from another tag collection. Scanning the full set of data items is time consuming but will be far less frequent than the selection of data items. We expect that there will only be 10 to 20 "base" tag collections in LHCb. All other collections will be subsets of base tag collections.

2.3.3 Selection Process

There is an immediate gain by selecting tags in tag collections instead of selecting directly data items. Only data items of interest are loaded instead of loading all items for each selection.

This is specially interesting in the case that data items are not located in a database but in regular files and loading a data item requires accessing a file containing many items. With a pointer to the data item within the file, the item of interest is directly accessed and loaded. Such a strategy of storing the actual data in regular files may actually be applied to many problems since database management systems cannot handle petabytes of data easily.

Furthermore, the 2-level indexing scheme presented here offers a very powerful and flexible way of applying various preselections allowing to reduce both the amount of accessed data and the network traffic. The complete selection process is shown in Figure 2.2.

The steps involved in the selection process are the following :

1. The client selects a tag collection and sends a SQL query to be applied on tags from this collection. The usage of a specific collection is actually a first preselection made by the physicist.
2. The query is processed on the server side.

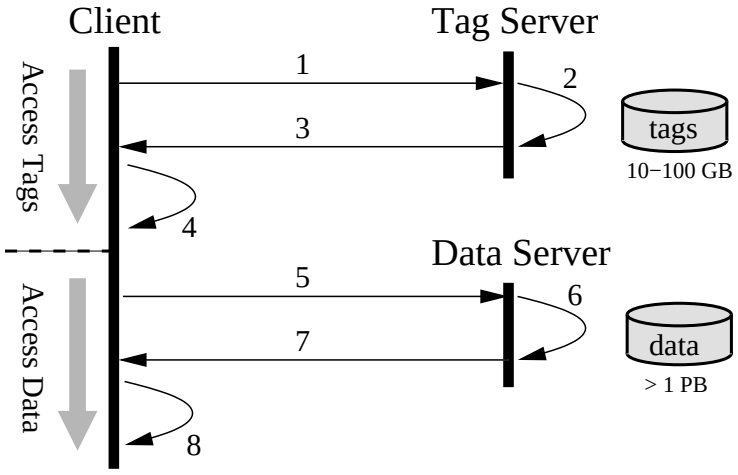

Figure 2.2: Data selection process

3. Only matching tags are sent back. This minimises the network load.
4. A second selection may be applied on the client side, for example for queries that cannot be formulated in SQL and which require a procedural language such as C++.
5. Once the selection on tags is complete, requests for the corresponding data items are sent to the data server.
6. Data items are retrieved (from files, in the LHCb experiment).
7. Retrieved data items are sent to the client.
8. A last selection may be performed on the full data items, in the case that some information was missing in the tags which did not allow to perform this more narrow selection in a previous step.

Note that the separation between client and servers (a tag server and a data item server) on Figure 2.2 allows for example to replicate the tag server while keeping the data item server at a single location .

2.4 Performance evaluation

Let us evaluate the performance of our indexing scheme. It is hard to compare our proposed scheme to existing indexing techniques since we don't know of other indexing techniques except linear scanning which are able to meet our requirements.

Two of the main high-dimensionality indexing schemes are X-trees [BKK96] and VA-file [WB97]. The X-Tree method is outperformed by a sequential scan for a dimension exceeding 6 (see [WSB98]) and VA-files are only applicable to similarity-search queries. Thus, we only compare our performances with the performances of the sequential scan method.

2.4.1 Some approximations

Let us make some simplifications and approximations in order to create a model of the proposed indexing scheme.

Type of data : We only consider one data type (integers). The cost of a comparison between two values is therefore always the same. This is not the case in real life, where data typically consist of numbers, booleans and strings. However, it is always possible to express the comparison cost of a data item type as a factor of a single integer comparison.

Optimisations : No optimisation of the query processing on tag collections are taken into account. This means that tag collections are searched sequentially. Thus, the gain obtained by querying tag collections is really the minimum we can expect from the new scheme.

Data transfers : No optimisation of data transfers are taken into account. Especially, we do not consider pipelined scheme where the data transfer of a given item could be carried out during the computation of the previous one.

Size of tag collections : For the performance analysis, we consider only a single tag collection with a fixed number of tags. The number of tags and the size of the tags may be considered as an average among the different values of a real life example.

Complex queries : We do not take into account complex queries that could only be processed by a dedicated program. In other words, step 4 of the selection process (Figure 2.2) does not occur here.

2.4.2 Theoretical model

Let us adopt the following notations :

- N is the number of items in the whole database;
- n is the average number of items in a given tag collection;
- D is the number of values in a data item i.e. its dimension;
- d is the number of values in a tag i.e. the dimension of the tag; we assume that all these values are tested;
- d' is the number of values that are not contained in the tag but still need to be tested (step 8 in Figure 2.2);
- T_{lat} is the latency of the network which is used to transfer the data;
- T_{tr} is the time used to transfer one value through the network; in second per value;
- T_{IO} is the time needed to load one value from disk into the memory of the Tag server or the Data server;
- T_{CPU} is the time to compute one value within the client node or within the tag server, i.e. the time to compare it with another value;
- q is the number of matching tags for the query we are dealing with;
- t_{seq} is the duration of the query using a sequential scan;
- t_{tag} is the duration of the query using the new indexing scheme.

In the case of a sequential scan, the time needed to process a query is simply the time needed for querying one data item multiplied by N. Each data item is read from disk, transferred through the network and processed.

$$t_{seq} = N\left(T_{lat} + D\left(T_{tr} + T_{IO}\right) + \left(d + d'\right)T_{CPU}\right)$$

It is independent of the size of the result.

The time needed to process a query using the new indexing scheme is slightly more complicated to derive. Using the architecture depicted in Figure 2.2, we can divide it into two parts : the duration t_q of the query on tags and the duration t_d of the query on data items. The query on tags is carried out on the tag server. Matching tags are transferred to the client. The query on data items is similar to the sequential scan method.

$$t_{tag} = t_q + t_d$$

$$t_q = n\left(d\,T_{IO} + d\,T_{CPU}\right) + q\left(T_{lat} + d\,T_{tr}\right)$$

$$t_d = q\left(T_{lat} + D\left(T_{IO} + T_{tr}\right) + d'\,T_{CPU}\right)$$

Finally :

$$t_{seq} = N\,T_{lat} + N\,D\,(T_{IO} + T_{tr}) + N\,(d + d')\,T_{CPU} \tag{2.1}$$

$$t_{tag} = 2\,q\,T_{lat} + (n\,d + q\,D)\,T_{IO} + q\,(d + D)\,T_{tr} + (n\,d + q\,d')\,T_{CPU} \tag{2.2}$$

The query duration is dependent on the number q of matching tags. Note that the assumption that tags are transferred one by one to the client corresponds to the worst case. This could be improved by sending groups of tags.

2.4.3 Interpretation

The terms in equations (2.1) and (2.2) can be divided into three parts : processing time (T_{CPU}), network transfer time (T_{lat} and T_{tr}) and data retrieval time (T_{IO}). Let us consider them separately.

Processing time : the processing time ratio between tag collection access and the default sequential scan is :

$$r_{CPU} = \frac{n\,d + q\,d'}{N\,(d + d')} = \alpha\,\frac{d + \gamma d'}{d + d'} \tag{2.3}$$

$$where \qquad \alpha = \frac{n}{N} \qquad \gamma = \frac{q}{n}$$

Since $\gamma \leq 1$ (comes from $q \leq n$), we can be sure that $r_{CPU} \leq \alpha$. This demonstrates that the CPU time ratio is less than (but of the order of) the ratio between the number of tags in a collection and the number of data items.

Network transfer time : the network transfer time ratio between tag collection access and the default sequential scan is :

$$\begin{aligned} r_{NET} &= \frac{2\,q\,T_{lat} + q\,(d + D)\,T_{tr}}{N\,T_{lat} + N\,D\,T_{tr}} \\ &= \frac{q}{N}\,\frac{2\,T_{lat} + (d + D)\,T_{tr}}{T_{lat} + D\,T_{tr}} \end{aligned}$$

Since $d \leq D$, we finally have :

$$\begin{aligned} r_{NET} &\leq 2\,\frac{q}{N} \\ r_{NET} &\leq 2\,\alpha\,\gamma \end{aligned} \tag{2.4}$$

$$where \qquad \alpha = \frac{n}{N} \qquad \gamma = \frac{q}{n}$$

Since $\gamma \leq 1$, the network transfer ratio is at least of the order of the ratio between the number of tags in a collection and the number of data items. In practise, we even have $\gamma \ll 1$ (we foresee $\gamma \sim 10^{-2}$ for LHCb) and thus $r_{NET} \ll \alpha$.

Data retrieval time : the data retrieval time ratio between tag collection access and the default sequential scan is :

$$r_{DR} = \frac{n\,d + q\,D}{N\,D} = \alpha\,(\beta + \gamma) \tag{2.5}$$

$$where \qquad \alpha = \frac{n}{N} \qquad \beta = \frac{d}{D} \qquad and \qquad \gamma = \frac{q}{n}$$

Usually, $\beta \ll 1$ and $\gamma \ll 1$. Thus $r_{DR} \ll \alpha$. This means that, in respect to data retrieval time, we gain far more than just the gain obtained by the preselection on data items.

Let us estimate γ. By definition, γ is the proportion of matching tags in a tag collection for a given query. Let us consider a very simple case where every part of the query is a comparison and is fulfilled by half of the items. In addition, let us suppose that the data is uniformly distributed. This leads to :

$$\gamma = \frac{1}{2^d} \qquad and \qquad \frac{r_{DR}}{\alpha} = \frac{d}{D} + \frac{1}{2^d}$$

Figure 2.3 gives the behaviour of this ratio against the dimension d for different values of D.

Roughly, $\frac{r_{DR}}{\alpha}$ goes down from 1 to a minimum for dimensions between 0 and $d_m \sim 8$ and increases linearly afterwards until it reaches 1 again for dimension D. Clearly, we can approximate r_{DR} by $\alpha\frac{d}{D}$ if $d \geq d_m$. This is exactly our goal since the data retrieval time becomes proportional to the loading time of the tags.

For the LHCb experiment, the dimension of a data item is typically $D \sim 20000$. The minimum I/O time is reached for $d \sim 18$ and $r_{DR} < \frac{\alpha}{1000}$.

2.5 Conclusions

We presented a new method of indexing and selecting data in huge datasets having a high index dimensionality. The method avoids linear scanning of the whole data set. Instead of scanning the whole dataset, we scan only the tags present in the

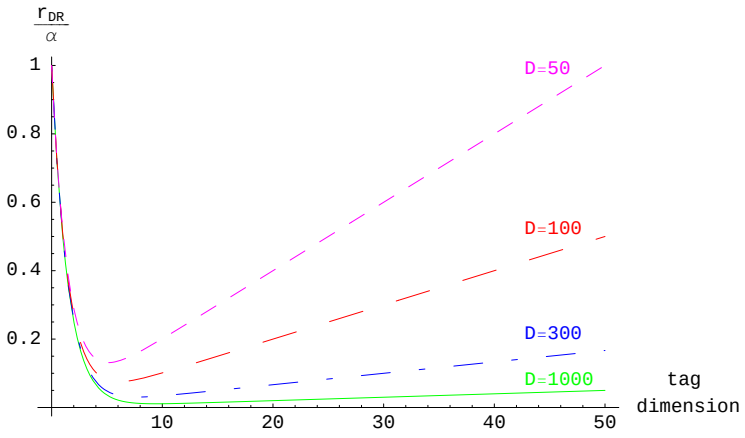

Figure 2.3: Evolution of a majoration of the data retrieval ratio divided by α in function of the dimension of the tag

tag collections. The selected tags point to the data items that are then retrieved for applying a more narrow selection.

By scanning tags in tag collections instead of a flat scan of all data items, the minimal gain is proportional to the ratio between the number of data items and the number of tags within the selected tag collections. In many cases, the effective gain is the minimal gain multiplied by the ratio of the dimension of the data items and the dimension of tags.

The proposed data item selection and retrieval scheme (see also [PVH02]) was implemented at CERN, in the context of the LHCb experiment and seems very promising.

Chapter 3

Parallelization policies

We propose scheduling policies for parallelizing LHCb analysis applications on computer clusters. LHCb analysis jobs require the analysis of tens of thousands of particle collision events, each event requiring typically 200ms processing time and 600KB of data. Many jobs are launched concurrently by a large number of physicists. At a first view, particle physics jobs seem to be easy to parallelize, since particle collision events can be processed independently one from another. However, since large amounts of data need to be accessed, the real challenge resides in making an efficient use of the underlying computing resources. We propose several job parallelization and scheduling policies aiming at reducing job processing times and at increasing the sustainable load of a cluster server. We demonstrate via cluster simulations how the different policies improve the system capabilities.

Section 3.1 presents the issues raised by the parallelization of LHCb analysis jobs. Section 3.2 presents search results in the domain of application parallelization. In Section 3.3, we introduce the simulation tools and the simulation parameters used to evaluate and compare the scheduling policies. In Section 3.4, we present simple 'first come first served' job parallelization and scheduling policies. In Section 3.5, we introduce out of order job scheduling and data replication policies. In Section 3.6, we propose a delayed scheduling policy optimising the resources of the system. Delayed scheduling may have disadvantages for end users but allows to sustain considerably heavier loads. We also propose an adaptive delay strategy that allows to minimise user waiting time at low and normal loads and to optimise the utilisation of the computing resources at high loads. In Section 3.7, we draw the conclusions.

3.1 Issues and plans

Most of the data are stored on tapes using Castor [CER05], a tertiary mass storage system manager developed at CERN that hides the tape archives from the user by streaming data and caching data on large disk arrays. Data retrieval time from tertiary mass storage is still about three times slower (i.e. typically 600ms for retrieving a 600KB data item) than the corresponding data processing time due to tertiary storage access and to the network throughput limitations.

The parallelization is facilitated by the nature of the processing patterns and of the corresponding data segments. Data segments comprise a succession of "small", fully independent collision events (around 600KB per event). Event processing consists of scanning and analysing the events one by one and of creating corresponding statistics. The event analysis output is very small. It comprises a set of histograms which can be easily merged when carried out in parallel on several nodes. Merging and transferring the results requires therefore a negligible effort. Since events can be analysed independently one from another (no data dependency), event analysis does not induce inter-node communications.

Particle collision events are usually reused by several jobs. We therefore present *cache based job splitting* strategies that considerably increase cluster utilisation and reduce job processing times. Compared with straightforward job scheduling on a processing farm, cache based 'first in first out' job splitting speeds up average response times by an order of magnitude and reduces job waiting times in the system's queues from hours to minutes.

By scheduling the jobs out of order, according to the availability of their collision events in the node disk caches, response times can be further reduced, especially at high loads. In the delayed scheduling policy, job requests are accumulated during a time period, divided into subjob requests according to a parameterizable subjob size, and scheduled at the beginning of the next time period according to the availability of their data segments within the disk node caches. Delayed scheduling sustains a load close to the maximal theoretically sustainable load of a cluster, but at the cost of longer average response times.

Finally we propose an adaptive delay scheduling approach, where the scheduling delay is adapted to the current load. This last scheduling approach sustains very high loads and offers low response times at normal loads.

3.2 Related Work

There are many strategies for scheduling parallel applications on a cluster of PCs [ZBWS98, AS97, KN93, AGR03]. However, these strategies are generally not applicable to data intensive applications which are arbitrarily divisible [BGR03], which access partly overlapping data segments and whose data needs to be loaded from tertiary storage. On the other hand, strategies focusing on data retrieval [ML96, BBS+94] deal mainly with I/O throughput without considering job distribution and scheduling issues. Kadayif et al. address the problem of data aware task scheduling but only for a unique CPU [KKKC02].

We present new paradigms that aim at optimising the parallelization and scheduling of jobs on a cluster of PCs for data-intensive applications. The proposed parallelization and scheduling policies rely on job splitting, disk data caching, data partitioning [TF98], out of order scheduling as well as on the concept of delayed scheduling.

3.3 Cluster simulations

Let us present here the cluster simulation tool that has been developed to validate the different job scheduling policies.

3.3.1 The framework

We needed a simulation framework allowing to develop and test different job parallelization and scheduling policies before running them on a real computing cluster. Since no efficient software adapted to our needs is available, we created our own simulation tools. Existing general purpose simulation tools [UKSS98, Gal99] simulate communications between nodes and are therefore too slow to simulate days or weeks of particle collision event processing.

Since particle collision event processing does not induce communications between computing nodes, we only take into account the communication time related to data transfers. The simulation framework (Fig. 3.1) simulates the behaviour of a cluster of PCs (CPU + memory + disk) connected via a high speed network (typically Gigabit Ethernet) to a shared tertiary storage device (e.g. the CASTOR [CER05] system at CERN).

Cluster management and job scheduling are carried out on a dedicated node called “master node”. The job parallelization and scheduling software may run

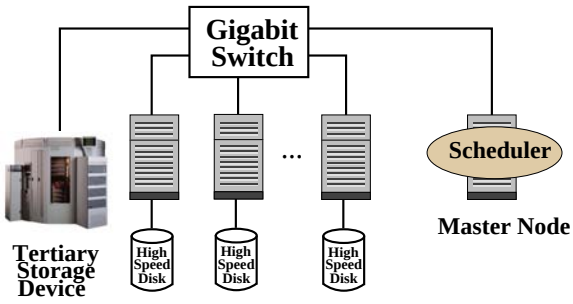

Figure 3.1: Architecture of the simulated cluster

both on the simulated and on the target system (production environment). It implements a plugin model, enabling new scheduling policies to be easily added.

Due to the available CPU power limitations, the simulations are carried out at a reduced scale. The total data space is reduced from 2 PB to 2 TB and the number of nodes in the cluster from 10000 to 10 or 20 nodes, in addition to the master node.

3.3.2 Parameters of the simulation

In real computing clusters, important parameters are CPU speed, node memory, disk space, disk throughput, etc. Within our simulation framework, we made the following assumptions :

- The processing node memory is considered to be infinite since we only run a single job or subjob per processor at any given time. We therefore expect the node memory to be always large enough.
- Processing nodes are single CPU computers and all nodes are identical.
- Processing power requirements are expressed in terms of single CPU seconds.
- Disk throughput is 10 MB/s.
- The node disk cache size is either 50 GB, 100GB or 200GB.
- The tertiary storage system is accessed through Castor [CER05]. This system caches tape data on disk arrays. We therefore do not take the tertiary storage system data access latency into account.
- Throughput from tertiary storage to each node is 1 MB/s.
- The total data space accessible by the high energy physics analysis jobs is 2 TB.

- The default number of processing nodes in the cluster is 10. Simulations were also carried out for 5 and 20 nodes and lead to similar results. Thus, we mostly present results for a cluster of 10 nodes.
- Jobs are typical high energy physics analysis jobs. They consist of a large collection of events, 40000 events on average. The number of events per job follows an Erlang probability distribution with parameter equal to 4 (see Figure 4.1 on page 62 for a graph of the Erlang distribution). Each event requires 200 ms CPU processing time and access to 600 KB of data.
- The events accessed by a given job are contiguous. The data segment they form starts at a random position within the dataspace. The distribution of the job start points within the dataspace is homogeneous except for two regions, representing together 10% of the total data space but incorporating 50% of the start points. Such a distribution mimics the fact that the fraction of the data associated with some very interesting events is accessed far more frequently than the remaining data. This start point distribution yields the data utilisation curve shown in Figure 3.2. The probability of having two jobs with overlapping data is around 6.7% (see Appendix B for details). Note that this probability would be 2.4% with a uniform distribution of start points (see Appendix B). When jobs have overlapping segments, the average size of the overlapping region is half the average size of the data segment associated to a job, i.e. 12 GB.
- Job requests arrive randomly, with an exponential distribution of intervals between arrival times. The mean number of arrival jobs per time (cadence) depends on the considered load.

3.3.3 Performance measurements

Two variables define the performance of a scheduling strategy, the *average waiting time* and the *average speedup*, both a function of the cluster *load*.

The *waiting time* is the time spent between job submission and beginning of job processing. It is interesting to compare it with the average time needed to run a simple isolated job on a single processing node without cache. In our context, the average single job single node processing time without disk cache is 32000 seconds, i.e. almost 9 hours.

The *speedup* of a job is the single job single node processing time without disk cache divided by the job processing time in the parallel system when scheduled according to the current policy. The *processing time* is defined as the time between

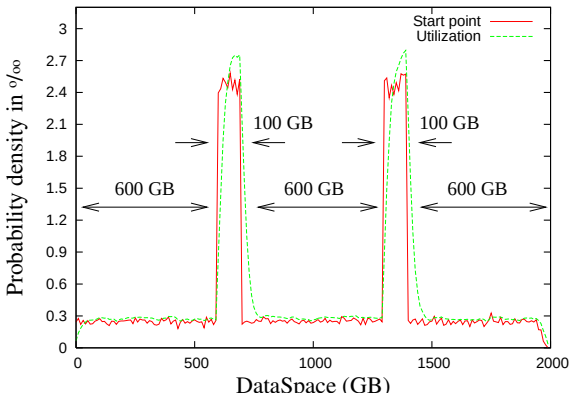

Figure 3.2: Start point and data utilisation distribution

the effective start of job processing, i.e. start of processing of the first part of the job, and the end of job processing, i.e. end of processing of the last part of the job. Thus, the processing time may include periods where the job or part of its subjobs are suspended.

The speedup is larger than one if there is a gain in terms of processing time. Two main factors contribute to the speedup : parallelization of jobs by job splitting and data caching. The parallelization is maximised when each job is subdivided into as many subjobs as available nodes. The performance improvement due to parallelization is thus lesser or equal to the number of nodes, i.e. 10 in our cluster configuration. Data caching is maximised when data is always read from disk caches instead of tertiary storage. In our context, the performance improvement due to maximal data caching is slightly larger than 3. Therefore the maximal overall speedup that can be reached is 30.

The *load* of the cluster is measured in terms of mean number of job arrivals per hour. The maximal load of a cluster corresponds to the load sustainable when all processors run at 100% CPU utilization and data is accessed from disk caches only. In our context (.26 s per event, 4000 events per job and 10 nodes), the maximal load is 3.46 jobs per hour.

Hereinafter, all measures make the assumption that the cluster runs in steady state. We do not take into account the startup period of the cluster, when empty disk caches are filled.

3.4 First come, first served scheduling policies

In this section, the proposed scheduling policies rely on the following basic principles :

- Once started, a job never hangs. Therefore, at least one dedicated node is allocated to it.
- Jobs are started in a first come first served order in order to ensure a fair treatment of user requests.

Keeping these principles in mind, we analyse the speedup that can be obtained with load balancing strategies such as simple job splitting and cache oriented job splitting.

3.4.1 Processing Farm oriented job scheduling

The simplest scheduling policy relies on the processing farm paradigm. This is the policy in use at CERN for scheduling jobs on a computing cluster comprising hundreds of nodes. Jobs are queued in front of the cluster and are transmitted to the first available node. This node remains dedicated to that job until its end. No disk caching is performed. All data segments are always transferred from tertiary storage when needed.

This simple scheduling algorithm is well studied and understood. A mathematical model can be established which describes the cluster behaviour as a special case of a M/Er/m queueing system [Kle76]. We simulate the processing farm oriented job scheduling policy as a reference for judging the performances of the proposed more advanced parallelization and scheduling policies.

3.4.2 Job splitting

Since jobs contain tens of thousands independent events, they may be split into subjobs running in parallel on different nodes of the cluster. Job splitting occurs when new nodes become available and would remain idle, i.e. no new jobs are in the queue. The job splitting scheduling policy ensures that the maximum possible number of nodes is used at any time. Data segments are only requested from tertiary storage when they are needed i.e. when the corresponding event analysis code is being executed. Job splitting induces therefore no data replication. The job splitting scheduling policy is described in Table 3.1.

Table 3.1: The job splitting scheduling policy

Upon *job arrival*

- If some nodes are idle, the new job is split into subjobs of equal sizes, one per idle node. All subjobs are launched in parallel. To avoid too small jobs, we do not split beyond a minimal job size (10 events).
- If no node is idle but some job(s) is/are running in parallel on several nodes, one node is released by the job having the largest number of nodes per event to process. The corresponding subjob is suspended and the new job is launched on the released node.
- If there are as many jobs running as nodes, the new job is queued.

Upon *subjob end* (but not job end)

- If there are suspended subjobs within the same job, one of them is activated and runs on the node becoming free.
- Otherwise the node is allocated to an already running job. The largest subjob running on the cluster is split into two equal parts, one of them being launched on the free node. Again, jobs below a minimal size are not split.

Upon *job end*

- If there are queued jobs, the first queued job is run.
- Otherwise the free node is allocated to an already running job, as in the case of *subjob end*.

The job splitting policy performs always better than the simple processing farm oriented job scheduling policy. The job processing time is always lower compared with the processing farm approach since there are always as many jobs running as in the processing farm approach and since job splitting reduces the job processing time. In both approaches, all data segments are transferred from tertiary storage.

3.4.3 Cache oriented job splitting

The job splitting scheduling policy remains very close to the processing farm approach since the disks of the processing nodes are not used for caching data. By always caching data arriving from tertiary storage onto node disks, we improve the effectiveness of job splitting. We try to schedule jobs on those nodes which

store, at least partly, their corresponding data segments. This strategy leads to the new cache oriented job splitting policy detailed in Table 3.2.

Table 3.2: Details of the cache oriented job splitting policy

Upon *job arrival*

- The new job is split into subjobs depending on the content of node disk caches : data processed by a given subjob should always either be fully cached on a node or not cached at all. Again, there is a lower limit on the smallest job size.
- If some nodes are idle, they are given the most suitable subjob (a fully cached subjob if such a subjob exists). If there are not enough subjobs for all nodes, they are further subdivided. If there are too many subjobs, the ones that cannot immediately be scheduled are suspended.
- If no node is idle but some job(s) is/are running in parallel on several nodes, one selected node is released by a selected job. Node and job selection are performed so as to maximise cached data access. We try to replace a subjob working with non cached data by the new job or one of its subjobs, working on cached data.
- If there are as many jobs running as nodes, the new job is queued.

Upon *subjob end* (but not job end)

- If there are suspended subjobs within the same job, one of them is activated on the node becoming free. The chosen subjob is the one having the largest amount of data cached on that node.
- Otherwise the node is allocated to an already running job. The subjob that is split is the one for which the caching benefit is the largest.

Upon *job end*

- If some jobs are in the queue, the first one is taken and run.
- Else if there are suspended subjobs, the most suitable one is activated.
- Otherwise an already running subjob is split, as carried out in case of *subjob end*.

The scheduler maintains the job and subjob queues as well as the state of all disk caches in the cluster. When needing new disk cache space, it deallocates the least recently used cached segments.

Cache oriented job splitting offers on average more performance than simple job splitting since it takes advantage of disk caches. Not every single job terminates earlier since the ordering may change but, in the general case, the average

time a job spends in the system is reduced. Furthermore, the cluster is better utilised and therefore capable of sustaining slightly higher loads.

3.4.4 Simulation Results

Figure 3.3 gives the average speedup and waiting time for different loads and for each of the scheduling policies described so far, including different cache sizes for the cache oriented job splitting. The curves on the graph are cut at high loads when the system leaves the steady state and becomes overloaded. When overloaded, the notion of average waiting time does not make sense anymore since jobs are accumulating and the waiting time grows to infinity.

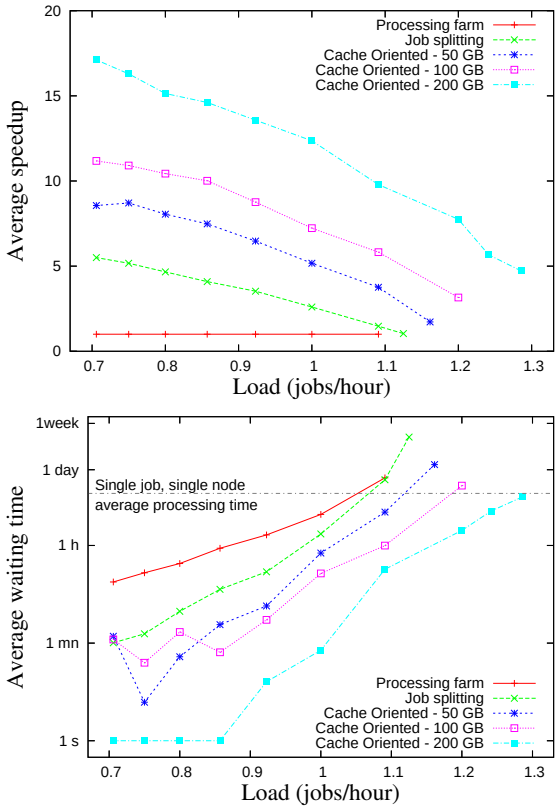

Figure 3.3: Average speedup and waiting time for different scheduling policies, different cache sizes and different load levels for a system comprising 10 processing nodes

The results show that the job splitting policy improves the performance, es-

pecially when the load is not too high. The cache size in the cache oriented job splitting policy appears to be decisive. The simulation shows that the maximal data caching speedup factor (i.e. 3) is reached for a disk cache size of 200GB (in Fig. 3.3, compare the job splitting with the cache oriented policy at low loads). For smaller caches, the gain in performance compared with the non cache-oriented job splitting is approximately proportional to the size of the disk cache.

Besides being vertically shifted, the waiting time curves are similar one to another. As foreseen, the policies providing a higher speedup induce a lower waiting time. Increasing the cache size decreases the job waiting time from days to hours.

3.5 Data distribution and data replication

In Section 3.4, we introduced a cache based job splitting policy ensuring a high degree of fairness by running the jobs in a first in first out order.

Let us study what can be gained by relaxing this constraint and analyse to what extend a certain degree of fairness can still be reached. Our hypothesis is that the usage of cache may be improved if we let jobs that find useful data in the cache execute before jobs that have to load their data from tertiary storage.

3.5.1 Out of order Scheduling

We define a new scheduling policy relying on out of order job scheduling that aims at making a maximal usage of node disk caches. Table 3.3 describes the out of order job scheduling policy.

This scheduling policy does not guarantee fairness i.e., according to the availability of cached data segments, the job execution order is modified. One may imagine a succession of jobs where a given job having no data in cache would never be executed. In order to ensure a minimal degree of fairness, we add an extra feature. Whenever the waiting time of a given job in the queue of jobs with no data cached exceeds a given maximum (2 days in our context), the job is run with a higher priority. The first available node executes this job before running any other job or subjob. When the cluster is not overloaded, such an event occurs very seldom since there will always be a time point at which the queues become empty and jobs with non cached data may be launched. The case where a job is assigned a higher priority typically occurs for less than 0.5 ‰ of the jobs.

Table 3.3: Details of the out of order job scheduling policy

Each node maintains a queue of subjobs. These subjobs only need data that is cached on their node. An extra queue contains subjobs with no cached data.
Upon job arrival

- When a job enters the cluster, it is split into subjobs so as to ensure that each subjob is either fully cached on a node or not cached at all. Jobs are not split beyond a minimal size.
- Subjobs with cached data are immediately run if the node is idle or if it is running a subjob without cached data. In that case, the former subjob is suspended and placed back at the first position of the queue where it came from (queue of subjobs with no cached data or a specific node queue).
- The remaining subjobs with cached data are queued on the nodes where their data is cached.
- If some nodes are still idle, they are fed with the subjobs having no cached data. These subjobs may be split in order to feed all nodes.
- Remaining subjobs with no data cached (if any) are put in the "no cached data" subjob queue.

Whenever one or several node(s) become(s) available

- If the node has subjob(s) waiting in its queue, the first subjob is run.
- Otherwise a subjob waiting in the "no cached data" subjob queue is run. In case several nodes are available and there are not enough subjobs in the queue, subjobs may be further split. The lower limit on the size of subjobs also applies here.
- If some nodes are still available (no subjobs at all in the special queue or too small ones to be split), they will take work from the most loaded nodes. By doing this, some subjobs that had cached data will be run on nodes that don't have the data. Thus, the subjobs are split so as to ensure that the two subjobs terminate around the same time. The new subjob incorporates a flag specifying that a subjob with cached data may take precedence over it in the future (see second point above).

This out of order job scheduling policy optimises the global job throughput by maximising accesses to data from disk caches. However, isolated jobs may have an exceptionally long waiting time.

Figure 3.4 gives the average speedup and waiting time for the out of order job scheduling policy. As in Figure 3.3, the curves are cut at high loads when the cluster becomes overloaded, i.e. when queues start growing indefinitely.

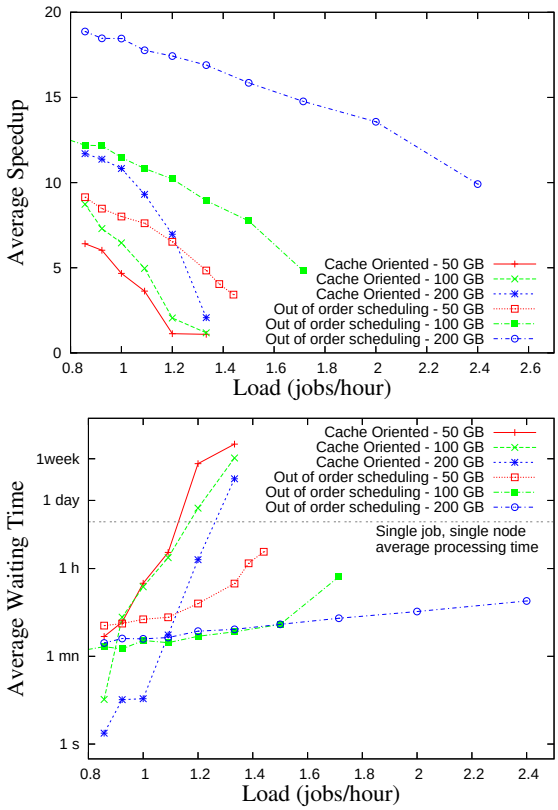

Figure 3.4: Average speedup and waiting time for cache-oriented job splitting and out of order scheduling policies

Figure 3.4 shows that the out of order scheduling policy performs on average much better than the cache oriented job splitting policy both from a cluster utilisation and a user point of view. For the same amount of cache and under the same load, we obtain a much higher speedup and an average waiting time which is an order of magnitude lower. The server also sustains far larger loads, especially in the case of large caches. The degradation of the speedup at the proximity of the maximal load is excellent.

Regarding the possibly longer waiting time for individual jobs, Figure 3.5 shows the typical waiting time distribution for the out of order scheduling policy near the maximal sustainable load. The worst-case waiting time is one to two days, depending on the cache size. This is acceptable since the single job single node average processing time is 9 hours. The waiting time distribution curves characterise the out of order scheduling policy. Arriving jobs can be classified

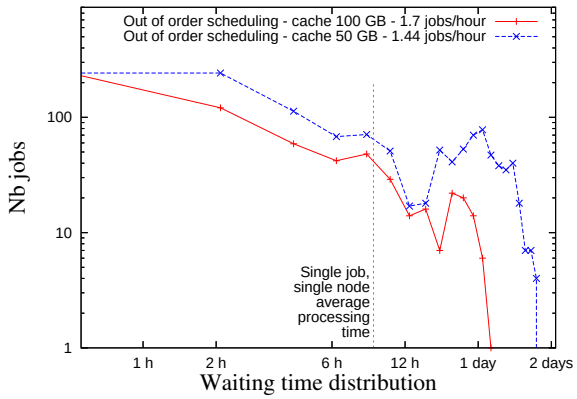

Figure 3.5: Waiting time distribution for the out of order scheduling policy near the maximal sustainable load.

into two categories : either they have cached data and can overpass the other jobs (left part of the curves) or they have no cached data and are overpassed (right part of the curves).

3.5.2 Data Replication

One may think that for high loads the previous scheduling algorithm may be further improved by performing data replication between the nodes of the cluster.

Whenever a node is overloaded and other nodes take work from it without having the corresponding data in their cache, it is pretty inefficient to grab the data again from the tertiary storage system. It is better to read the data directly from the disk of the overloaded node and copy it onto the local node disk.

However, such a replication strategy has drawbacks. The replication of the data segment onto the new node requires another data segment to be removed from the disk cache. The removed data segment may have been useful for future jobs. Thus, replicating a data segment when using the replicated data segment only once is not worthwhile. It is therefore preferable to directly read the data segment from the other node and use it without replication.

Replication should take place when the cost of not replicating is larger than the cost of replication [BFR92, FS00]. Applying this principle, we adopted the following strategy. Whenever a node A works on a data segment that is cached on another node B, the data segment is remotely read from B. We assume that B does not slow down due to the activity implied by this access. We also assume

that B is able to send the segment to A as soon as its disk becomes idle, i.e. in parallel with the processing of its current job (pipelining of disk I/O and CPU).

By default, a data segment read from a remote node is not put in the cache of the new node. A data segment is replicated only when the cost of not having replicated it from the beginning exceeds the cost of the replication. This information is obtained by keeping in each node the number of remote accesses to its data segments. In our context, data replication is carried out only on data items that are accessed for the third time.

Out of order job scheduling with and without data replication are compared in Figure 3.6. Basically, there is no difference : data replication does not improve our scheduling algorithm.

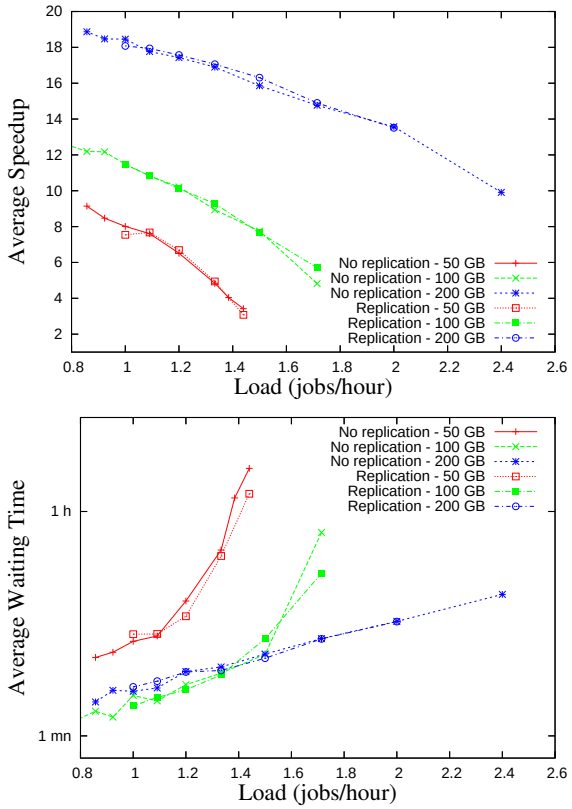

Figure 3.6: Gain in speedup and waiting time by considering replication versus no replication of data items for the job out of order scheduling algorithm

The reason for the poor behaviour of data replication appears clearly if one tries to analyse the consequences of the out of order scheduling policy in respect

to the distribution of data across the different nodes. The scheduling algorithm, taking advantage of job or subjob independence, always tries to use all available nodes whenever a job arrives. This leads to a situation where jobs are always split onto many nodes. As an example, the first job of each busy period will be split onto all nodes. Even jobs starting on a single node will, after some time, take advantage of the nodes released by other terminated jobs.

Therefore, there is never a large continuous segment of data residing within a single node disk cache. A large data segment is always split onto several nodes. When a new job requiring a large data segment arrives, it will be immediately split onto several nodes, ensuring a high degree of load balancing.

The case where a node is overloaded compared with other nodes, i.e. where data segment replication would improve the performance, occurs very seldom. The detailed analysis of the simulation reveals that data replication is used in less than 1 ‰ of the job arrivals.

3.6 Towards cluster-optimal scheduling

We consider a policy to be optimal from a cluster utilisation point of view if it is able to sustain a larger load than any other policy. In our context, the load is defined as the mean number of job arrivals per time interval, all other conditions being identical.

With this definition of optimality, one can enumerate the properties of an optimal scheduling policy. The maximal load occurs when all data is cached and all nodes are fully utilised. This however can never happen if the total disk cache in all nodes is less than the total data size. Thus an optimum can be achieved if data is loaded at most once from tertiary storage. An optimal behaviour supposes that the scheduler has a complete knowledge of all future jobs in order to schedule jobs before their data segments are removed from the cache. In other words, an optimal policy is an offline policy.

3.6.1 Delayed scheduling

The previous statement does not help much in an on-line context. But it shows a tendency : the more we know about future jobs, the better we can schedule them.

This leads to the definition of a *delayed scheduling policy* where several jobs are scheduled at fixed time intervals. Time is divided into periods of equal size

Table 3.4: Details of the delayed scheduling policy

Each node maintains a queue of subjobs. These subjobs only need data that is cached on their node. An extra queue contains meta-subjobs with no cached data. A meta-subjob is an aggregation of subjobs requiring overlapping data segments.
At the end of a period, all waiting jobs need to be scheduled. Jobs are scheduled as follow :

- Each job is split into subjobs so as to ensure that each subjob data segment is either fully cached on a node or not cached at all. There is a lower limit on the subjob size.
- Subjobs with cached data are queued on the corresponding nodes.
- The other non-cached subjobs are further split as follows :
  - A list defining data segment start and end points of subjobs is built
  - Points creating stripes below half the "stripe size" are removed. Points are also added so as to ensure that no stripe is above the "stripe size"
  - The final list of points is used to split the subjobs into subjobs having a number of events equal to or lower than the stripe size.
- Non cached subjobs working on the overlapping data segments are gathered into meta-subjobs. Note that the size of the meta-subjob data segments is defined by the stripe size.
- Meta-subjobs are queued according to their arrival time so as to introduce fairness in their order of execution. The arrival time of a meta-subjob is defined as the earliest of its subjobs arrival times.

Once the queues are filled, a new period starts during which the subjobs are processed as follow :

- Nodes run in priority the subjobs located in their private queue.
- If a node's queue is empty and the node becomes idle, it pops the first meta-subjob from the queue of non cached meta-subjobs and places every subjob contained in it into its queue. By construction, these subjobs are all requiring a partially common contiguous data segment which is not present on the node and which will be loaded from tertiary storage.

during which jobs are accumulated without being scheduled. They are then scheduled at once at the end of the period and processed during the next period. A data segment *stripe size* is also defined as being the largest acceptable size of a data segment associated to a subjob. We use different values for the stripe size, ranging

from 200 to 25000 events. Table 3.4 describes the delayed scheduling policy.

The goal of the delayed scheduling policy consists in loading the data from tertiary storage only once during a given period. In addition, the "stripe size" parameter controls the average size of a subjob and thus on how many nodes a job may be distributed.

The comparison between the out of order and the delayed scheduling policies is not obvious since they don't try to achieve the same goal. The out of order scheduling policy, despite its name, is still pretty fair concerning the job execution order. On the contrary, the delayed scheduling policy only focuses on cluster utilisation optimisation. The average speedup will thus be lower since many jobs requiring non cached data may stay idle during long periods while other jobs are running on cached data (no fairness). For the same reason, their waiting time will also be worse. In addition the waiting time becomes longer due to the fact that jobs have to wait until the end of a period before being scheduled. This extra "period" delay is not included in the waiting time shown in the figures.

Figure 3.7 on page 55 shows the results of the simulation for delayed scheduling with different "period" delays. As in Figure 3.3, the curves in Figure 3.7 are cut at high loads when the cluster becomes overloaded i.e. when queues start growing indefinitely. For comparison purposes, the out of order scheduling policy is also shown. Delayed scheduling behaves poorly both in terms of average speedup and average waiting time. On the other hand, it allows to sustain very high loads, especially if the "period" delay is large (up to 1 week for 9 h jobs). However the total waiting time becomes really high if one takes into account the "period" delay.

The influence of the stripe size on the delayed scheduling algorithm performance is presented in Figure 3.8. It shows a very clear improvement in term of speedup for small striping values and only a negligible influence on the average waiting time. This reinforces the idea that the parallelization potential is better exploited with smaller stripe sizes. A larger average speedup allows to sustain higher loads.

A summary of the maximal sustainable loads under different period delays, cache sizes and stripe sizes is given in Figure 3.9. We can observe an almost linear dependency of the maximal load with respect to both the delay and the stripe size. The more we wait and the finer we split the jobs, the larger the maximal load we can sustain. The simulation experiments show that a maximal load of 3 jobs per hour can be reached by using 200 gigabytes of disk cache, 1 week of delay and a stripe size of 200 events. This maximal load can be compared with the maximal theoretical load of 3.46 jobs per hour. It is close to 3 times the load of 1.1 jobs per

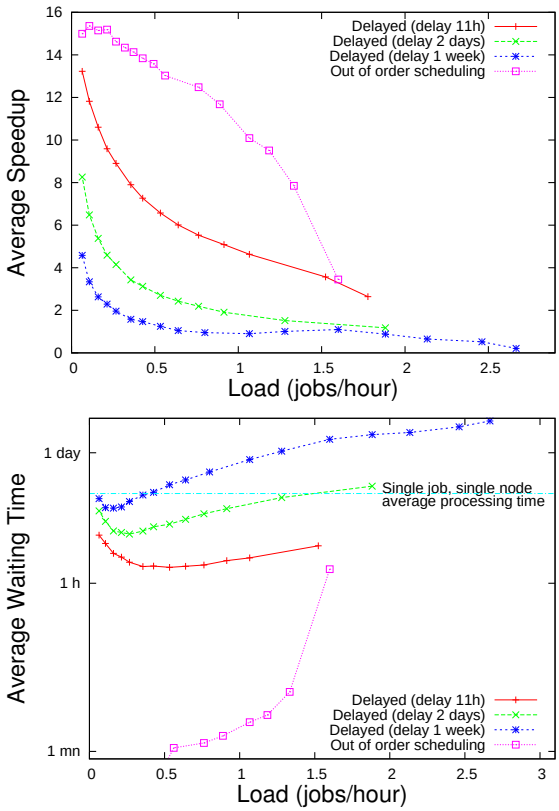

Figure 3.7: Speedup and waiting time (period delay excluded) of the delayed scheduling policy for different delays (cache size 100 GB, stripe size 5000 events)

hour sustained by processing farm scheduling without disk caching (see Sections 3.4.1 and 3.4.4).

3.6.2 Adaptive delay scheduling

As shown in the previous section, large period delays allow to sustain much higher loads at the cost of excessive waiting times for the end-users.

We define here a new adaptive delay policy that aims at minimizing the waiting time, while sustaining the current load. This policy makes use of the performance parameters shown in Figures 3.7, 3.8 and 3.9 in order to choose the minimal "period" delay that allows to sustain the current load.

Figure 3.10 shows the performance of adaptive delay scheduling compared to

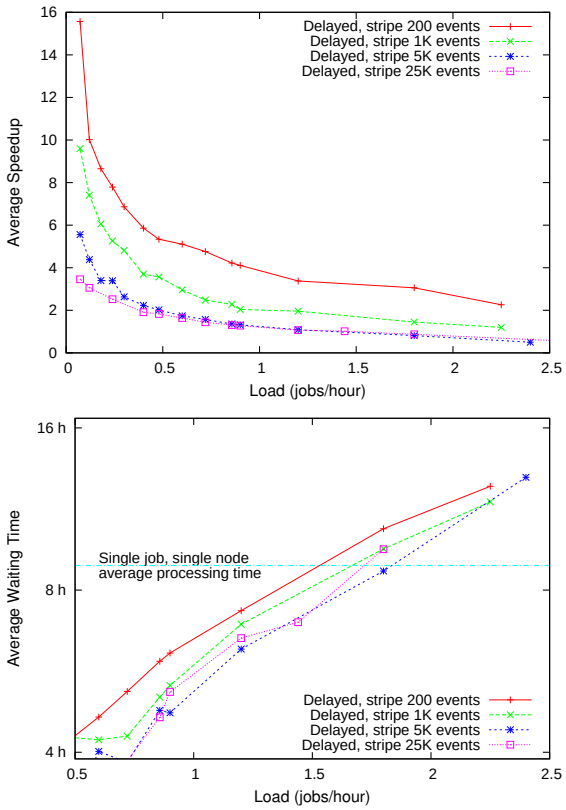

Figure 3.8: Average speedup and waiting time (delay excluded) of delayed scheduling for different stripe sizes (cache size 100 GB, period delay 2 days)

the out of order scheduling policy. As in previous figures, the curves are cut at high loads when the cluster becomes overloaded. As expected, the use of delayed scheduling allows the adaptive delay policy to sustain approximately 50% higher loads compared with the out of order scheduling policy.

At low loads and for small stripe sizes, the adaptive delay policy is performing in terms of speedup as well or slightly better than the out of order scheduling policy. At these low loads, the "period" delay is actually reduced to zero. The counterpart is a little overhead (up to 1h) in the average waiting time. However, this overhead is not really significant when compared to the single job single node average processing time (9h).

One may wonder why there is still a difference between the out of order scheduling policy and the delayed policy at low loads (where the "period" delay is zero). This is a consequence of the different data distribution strategies.

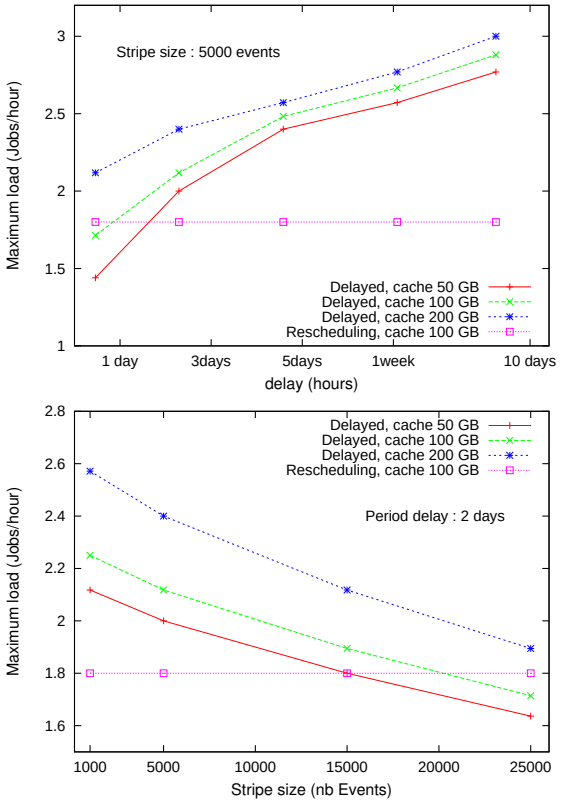

Figure 3.9: Maximal sustainable load of the delayed scheduling policy with different cache sizes in function of the period delay and the stripe size (stripe size on the left : 5000 events, period delay on the right : 2 days)

In the out of order scheduling policy, the data distribution is a side-effect of the parallelization that occurs when computing nodes are idle. However priority is given to starting new jobs over parallelizing (splitting) running jobs. In the delayed scheduling policy, in most of the cases, the data distribution is triggered by a predefined stripe size. Thus, for small stripe sizes, the level of parallelization of delayed scheduling is higher, leading to larger speedups. On the other hand, since jobs are in most cases split into subjobs running in parallel, and since only one subjob runs on one node at a given time, the number of concurrently running jobs becomes smaller, leading to longer waiting times.

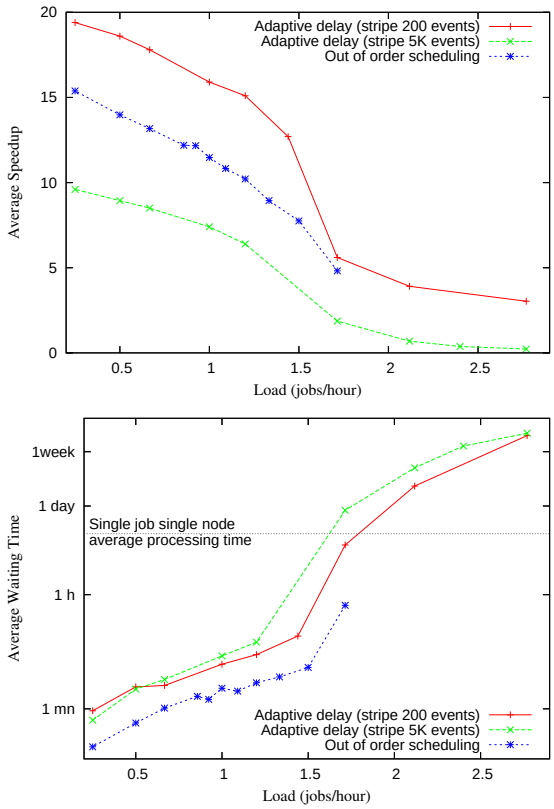

Figure 3.10: Speedup and waiting time (delay included) of the adaptive delay policy for different stripe sizes (cache 100 GB) compared with the out of order scheduling policy

3.7 Conclusions

We propose several scheduling policies for parallelizing data intensive particle physics applications on clusters of PCs (see also [PH04]) . Particle physics analysis jobs are composed of independent subjobs which process either non-overlapping or partly overlapping data segments. Jobs comprise tens of thousands of collision events, each one requiring typically 200 ms CPU processing time and access to 600 KB of data.

We show that splitting jobs into subjobs improves the processing farm model by making use of intra-job parallelism. By caching data on the processing farm node disks, cached-based job splitting further improves the performances.

The out of order job scheduling policy we introduce takes advantage of cache

resident data segments and still includes a certain degree of fairness. It offers considerable improvements in terms of processing speedup, response time and sustainable loads. For the same level of performance, the typical load sustainable by the out of order job scheduling policy is double the load sustainable by a simple first in first out cache-based job splitting scheduling policy.

We propose the concept of delayed scheduling, where the deliberate inclusion of period delays further improves the disk cache access rate and therefore enables a better utilisation of the cluster. This strategy is very efficient in terms of the maximal sustainable load (50 to 100% increase) but behaves poorly in terms of response time and processing speedup. In order to offer a trade-off between maximal sustainable load and response time, we introduce an adaptive delay scheduling policy with large delays at high loads and zero delays at normal loads. The delay is adapted to the current system load, thus trying to optimise the response time as a function of the current load. This adaptive delay scheduling policy aims at satisfying the end user whenever possible and at the same time enables sustaining high loads.

Chapter 4

Theoretical models

In this chapter, we try to build theoretical models of different scheduling policies in order to derive equations describing the system. We develop equations for the speedup for the job splitting scheduling policy, the out of order scheduling policy and the delayed scheduling policy. However, the cache oriented scheduling policy was not modelled due to the high complexity of its queuing system.

We also evaluate the scalability of the scheduling system depending on the complexity of the algorithms both in term of scheduling time and memory usage.

4.1 Assumptions and notations

The assumptions and notations given in this section are the common base for all the models developed in this chapter.

4.1.1 Job arrivals

Since the different jobs are issued by independent users, the proposed theoretical models suppose that the system has no memory about jobs arrival. This leads to an exponential distribution of the delay between two arrivals as shown in [Kle76, Section 2.4]. We call λ the parameter of this distribution. After a job arrival, the distribution of the delay before the next job arrival is thus :

$$P[\text{time before next arrival} < t] = e^{-\lambda\, t} \tag{4.1}$$

where λ also expresses the average number of jobs arriving per unit of time.

4.1.2 Service time

The distribution of the service time on a single node is supposed to be Erlangian with parameter r and mean service time $\frac{1}{\mu}$. The distribution of the service time is :

$$p[t \leq service\ time < t + h] = \frac{(r\ \mu)^r\ t^{r-1}\ e^{-r\ \mu\ t}}{\mu\ (r-1)!} h \tag{4.2}$$

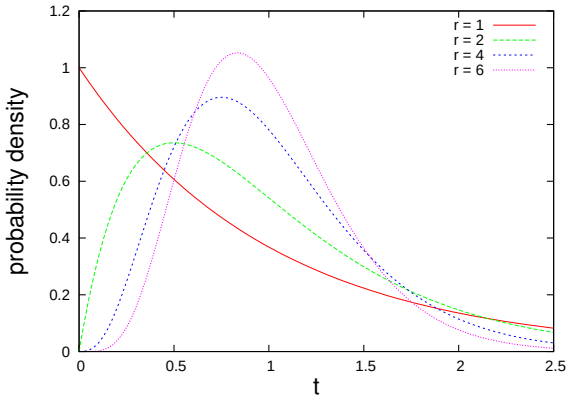

Figure 4.1: Erlangian probability Distribution with μ = 1

The choice of this distribution is a trade-off between easyness of the computations and the proximity of the distribution to the real case. The Erlangian distribution enables fixing the mean value of the distribution ($\frac{1}{\mu}$) and its correlation ($\frac{r}{\mu}$) independently. Figure 4.1 shows how the correlation factor affects the shape of the probability density. It also shows that for $r > 2$, the shape is reasonably close to a real case (usually a Gaussian shape). On the other hand, the Erlangian distribution is close to the very simple case of the exponential distribution (case $r = 1$) and thus enables simple computations.

An Erlangian distribution corresponds to a system where a user request has to go through r consecutive sequential servers, all identical and having each one an exponential service time distribution of parameter $r\ \mu$. The next user request can only enter the system when the first one is out of the last server. In other words, at most one of the r servers is active at any point in time. The global service time distribution of this system is Erlangian with average service time $\frac{1}{\mu}$ and parameter r.

This particularity can be used to reformulate the initial problem into a new one where each job or subjob has to pass through r service steps (the r servers in

the previous description), all having an exponential service time distribution. For more details, please refer to [Kle76, Section 4.3], concerning the $M/E_r/1$ queue.

4.1.3 Load

We call ρ the "load" of the system and define it as the ratio of job arrival rate and job service rate :

$$\rho \triangleq \frac{\lambda}{n\,\mu} \tag{4.3}$$

where n is the number of nodes in the cluster.

4.2 Job splitting Model

In the job splitting policy, each job can be split into several subjobs with no restriction on the number of subjobs and the subjobs sizes. In order to model this behaviour, we will first consider a case where each job is split into exactly m subjobs of equal size. We will then consider the limit when $m \to +\infty$ to mimic the real case. By taking this limit, we reach the case of an infinite number of subjobs. The scheduling will end up in grouping these subjobs into a finite number of subjob sets that match exactly the subjobs that would have been created in the real case. Thus the model will give the same average service time and speedup as the real scheduling strategy when $m \to +\infty$.

As explained in Section 4.1.2, we assume an Erlangian distribution of the service time by assuming that the service is divided into r sequential equal service steps having each an exponential service time distribution.

Let $p_i(t)$ denote the probability that the subjobs in the system at time t need to perform i service steps before finishing. We can estimate the evolution of this quantity after a very short duration h. Since h is short, we only take into account the cases when a single event takes place during h. There are three cases :

- no subjob go to the next service step and no new job arrives.
- one subjob goes to next service step and no new job arrives.
- a new job arrives and no subjob goes to next service step. Note that this job will be split into m subjobs according to our model and will perform $m\ r$ steps in total.

According to the job arrival time distribution (see Section 4.1.1), the probability that a new job arrives within h is $\lambda\, h$. According to the service time distribution (see Section 4.1.2), the probability that a given subjob finishes a given service step on a given node within h is $r\ \mu\ h$ (in case of a non idle cluster only). Therefore the probability that a step is finished by any subjob on any node within h is $n\ m\ r\ \mu\ h$ where n is the number of nodes and m is the number of subjobs. The sequencing of steps with the different transition probabilities is summarised in Figure 4.2.

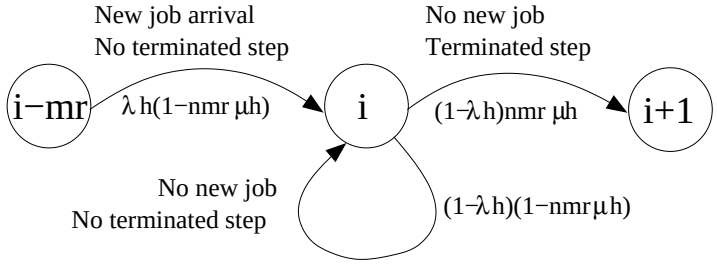

Figure 4.2: Sequence of steps for job splitting

If we take by convention $i < 0 \Rightarrow \forall t\ p_i(t) = 0$, this leads to the following equations for the evolution of the system :

$$\begin{aligned} p_0(t+h) &= p_0(t)(1-\lambda\ h) && \text{; Probability to remain empty (no new job)} \\ &+ p_1(t)(1-\lambda\ h) n\ m\ r\ \mu\ h && \text{; Probability to terminate last step} \end{aligned}$$

and $\forall i > 0$

$$\begin{aligned} p_i(t+h) &= p_i(t)(1-\lambda\ h)(1-n\ m\ r\ \mu\ h) && \text{; No new event, no termination} \\ &+ p_{i+1}(t)(1-\lambda\ h) n\ m\ r\ \mu\ h && \text{; No new event, end of step} \\ &+ p_{i-mr}(t)\lambda\ h(1-n\ m\ r\ \mu\ h) && \text{; New job, no termination} \end{aligned}$$

Neglecting the terms containing h^2, this leads to :

$$\frac{p_0(t+h)-p_0(t)}{h} = n\ m\ r\ \mu p_1(t) - \lambda p_0(t) \tag{4.4}$$

and $\forall i > 0$

$$\begin{aligned} \frac{p_i(t+h)-p_i(t)}{h} &= (-\lambda - n\ m\ r\ \mu)\ p_i(t) \\ &+ n\ m\ r\ \mu\ p_{i+1}(t) \\ &+ \lambda\ p_{i-mr}(t) \end{aligned} \tag{4.5}$$

Since we want to analyse the system in stable mode, the solution to equations (4.4) and (4.5) must be independent of time. We thus drop the time dependence and make the left term equal to 0 which leads to the following equation for p_i :

$$p_1 = \frac{\rho}{m\,r} p_0(t) \tag{4.6}$$

and $\forall i > 0$

$$p_{i+1} + \frac{\rho}{m\,r} p_{i-mr} = \left(1 + \frac{\rho}{m\,r}\right)\, p_i \tag{4.7}$$

Let us use the z-transform [Vic87] to solve these equations. Thus we define $P(z)$:

$$P(z) = \sum_{i=0}^{+\infty} p_i\, z^i \tag{4.8}$$

From equations (4.6) and (4.7) we get :

$$\sum_{i=1}^{+\infty} p_{i+1}\, z^i + \frac{\rho}{m\,r} \sum_{i=1}^{+\infty} p_{i-mr}\, z^i = \left(1 + \frac{\rho}{m\,r}\right) \sum_{i=1}^{+\infty} p_i\, z^i$$

$$\frac{1}{z}\left(P(z) - z\,p_1 - p_0\right) + \frac{\rho}{m\,r} z^{mr}\, P(z) = \left(1 + \frac{\rho}{m\,r}\right)\left(P(z) - p_0\right)$$

$$P(z) = p_0 \frac{1}{1 - \frac{\rho}{m\,r} z\, A(z)} \tag{4.9}$$
$$\text{with } A(z) = \frac{z^{mr} - 1}{z - 1}$$

Using the definition of $P(z)$, we have $P(1) = \sum_{i=0}^{+\infty} p_i = 1$. Using equation 4.9, we also have :

$$P(1) = p_0 \frac{1}{1 - \frac{\rho}{m\,r} z\, A(1)}$$

Since $A(1) = mr$ (see Appendix A), we can deduce that $p_0 = 1 - \rho$.

$P(z)$ gives every information on the behaviour of the system. In particular, we can derive N_m, the average number of jobs in the system. Since the average number of service steps in the system is $\sum_{i=1}^{+\infty} i\; p_i$ and the average number of remaining service steps per job is $\frac{m\,r}{2}$, we have :

$$N_m = \frac{2}{m\,r} \sum_{i=1}^{+\infty} i\,p_i \tag{4.10}$$

From equations (4.9) and (4.10) we get :

$$\begin{aligned} N_m &= \frac{2}{m\,r} \left.\frac{\partial P(z)}{\partial z}\right|_{z=1} \\ &= \left.\frac{2\rho(1-\rho)}{m^2\,r^2} \frac{A(z) + z\frac{\partial A(z)}{\partial z}}{\left(1 - \frac{\rho}{m} z\,A(z)\right)^2}\right|_{z=1} \end{aligned}$$

Since $A(1) = mr$ and $\left.\frac{\partial A(z)}{z}\right|_{z=1} = \frac{m\,r(m\,r-1)}{2}$, we finally have :

$$\begin{aligned} N_m &= \frac{2\rho(1-\rho)}{m^2\,r^2} \frac{m\,r + \frac{m\,r(m\,r-1)}{2}}{(1-\rho)^2} \\ N_m &= \frac{m\,r+1}{m\,r} \frac{\rho}{1-\rho} \end{aligned} \tag{4.11}$$

Finally, when $m \to +\infty$, we obtain the average number of jobs in the real case :

$$N_\infty = \frac{\rho}{1-\rho} \tag{4.12}$$

Note that this result is identical to the case of a $M/M/1$ system where a single node has an exponential service time distribution [Kle76].

The average service time AST for a job in the job splitting policy can be evaluated from N_∞. The average service time if the number of jobs in the system would always be equal to i would be :

$$AST_i = \begin{cases} \frac{i}{n\,\mu} & \text{if } i \leq n, \\ \frac{1}{\mu} & \text{if } i \geq n, \end{cases} \tag{4.13}$$

With P_i, the probability that there are exactly i jobs in the system, we have :

$$AST = \frac{\sum_{i=1}^{+\infty} P_i\,AST_i}{\sum_{i=1}^{+\infty} P_i} = \frac{\sum_{i=1}^{+\infty} P_i\,AST_i}{1 - P_0} \tag{4.14}$$

In the case of a sustainable load, the probability that the number of running jobs is greater than the number of nodes is very small. In other words, $\forall\ i > n\ \ P_i \ll 1$. We can thus evaluate AST by writing :

$$\begin{aligned} AST &\approx \frac{\sum_{i=1}^{+\infty} P_i \frac{i}{n\,\mu}}{1 - P_0} \\ AST &\approx \frac{1}{n\,\mu\,(1 - P_0)} \sum_{i=1}^{+\infty} i\,P_i \end{aligned} \tag{4.15}$$

By construction, the sum in the second term of equation (4.15) is the average number of jobs in the system, i.e. N_∞. Noting that $P_0 = p_0 = 1 - \rho$, and using equation (4.12) we have :

$$\begin{aligned} AST &\approx \frac{N_\infty}{n\,\mu\,(1 - p_0)} \\ AST &\approx \frac{1}{n\,\mu\,(1 - \rho)} \end{aligned} \tag{4.16}$$

We then derive the speedup, i.e. the relative decrease of the average service time, in respect to the case of single job running on a single node (case $\rho = 0$, $n = 1$) :

$$\begin{aligned} Speedup &\triangleq \frac{1}{\mu\,AST} \\ Speedup &\approx n\,(1 - \rho) \end{aligned} \tag{4.17}$$

Figure 4.3 shows the accuracy of the theoretical equations compared to the simulation.

4.3 Out of order Scheduling Model

4.3.1 Speedup derivation based on Job Splitting case

In the out of order scheduling policy, the main variation compared to the job splitting policy is the presence of the disk data caches and the way subjobs are scheduled depending on whether they access cached data or non cached data. In the case where the data caches have a zero size, the subjobs are not reordered and the two policies are similar. Thus, as a first approximation, the equations (4.12), (4.16) and (4.17) apply.

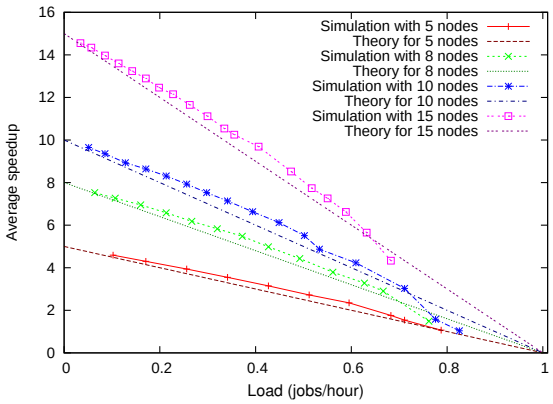

Figure 4.3: Average speedup time for the job splitting policy compared with the theoretical estimation

In the case of non-zero size disk caches, the effect of the caches can be seen as a modification of the average service time of each node. Since the subjobs can be reordered, the out of order policy will most of the time be able to take advantage of cached data. This is actually the main difference with the cache oriented job splitting policy where the subjobs are not reordered and very often have to be started without taking cached data into account.

Let γ be the gain in processing time when a node accesses cached data compared to its accesses to non cached data. We use here the following parameters, corresponding to the setup of the simulation in Section 3.3.2 :

- Disk throughput : 10 MB/s,
- Tertiary storage throughput : 1 MB/s
- Average event size : 600 KB
- Average CPU time per event : 200 ms

By assuming that the out of order policy manages to always use cached data by reordering subjobs, we have :

$$\begin{aligned} \gamma &= \frac{T_{I/O\ 3^{rd}\ storage} + T_{CPU}}{T_{I/O\ disk} + T_{CPU}} \\ &= \frac{0.6s + 0.2s}{0.06s + 0.2s} \\ &\approx 3 \end{aligned} \tag{4.18}$$

This gain γ is only achieved on cached data, otherwise the gain is 1. The average gain Γ depends on the probability that a given event is cached. It depends therefore on the size of the disk caches. Let c be the probability that a given event is cached on a given node. Thus $n\ c$ is the probability a given event is cached, assuming that caches contain distinct events. c is derived from Figure 3.2 on page 42 using the equations given in Appendix B on page 111.

We obtain :

$$\begin{aligned} \Gamma &= P[\text{data is cached}]\ \gamma + P[\text{data is not cached}] \\ &= n\ c\ \gamma + 1 - n\ c \\ &= 1 + n\ c\ (\gamma - 1) \end{aligned} \tag{4.19}$$

Let us now reconsider Section 4.2 with a single node service rate $\mu' = \Gamma\ \mu$, which implies a load of $\rho' = \frac{\rho}{\Gamma}$. This leads to an average service time :

$$AST \approx \frac{1}{n\ \mu\ \Gamma\ \left(1 - \frac{\rho}{\Gamma}\right)} \tag{4.20}$$

and therefore :

$$Speedup \approx n\ (\Gamma - \rho) \tag{4.21}$$

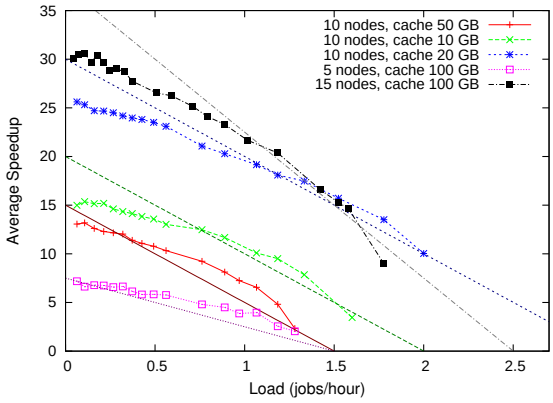

Figure 4.4: Average speedup time for out of order scheduling policy compared with the theoretical estimations

Figure 4.4 compares these theoretical estimations with the simulation results. Although the equations give an approximation of the simulated speedup, the shape of the speedup evolution with respect to the load is not correct. This is probably

due the fact that in the out of order scheduling policy, the reordering of job requests has not been modeled.

At high load, the reordering makes the out of order scheduling policy closer to the delayed scheduling policy than to the job splitting scheduling policy. This is due to the fact that jobs working on non cached segments will be delayed until all the jobs with cached segments are processed.

4.3.2 Improved model

Figure 4.4 shows that the approximation given by equations 4.20 and 4.21 are too rough. The simulations tend to suggest that the actual equations for the average service time and the speedup should be :

$$\Gamma = 1 + \frac{3}{4} n\, c\, (\gamma - 1) \tag{4.22}$$

$$AST \approx \frac{1}{n\, \mu\, \Gamma \sqrt{1 - \frac{\rho}{\Gamma}}} \tag{4.23}$$

$$Speedup \approx n \sqrt{\Gamma} \sqrt{\Gamma - \rho} \tag{4.24}$$

Figure 4.5 gives a comparison of the simulations and the theory with this set of empirically derived equations.

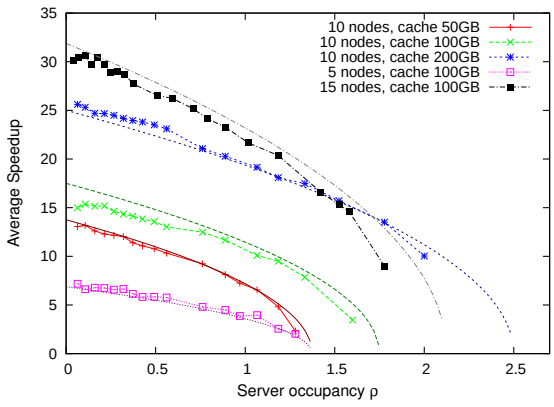

Figure 4.5: Average speedup time for out of order scheduling policy compared with the improved equations

4.3.3 Number of subjobs per jobs

The number of subjobs created per job is an important parameter of the system and is needed to compute the algorithms complexities (Section 4.5). Therefore, we need to establish the dependency between the number of subjobs per job and the number of nodes in the cluster.

Let $j(t)$ be the number of jobs running concurrently in the cluster and $s(t)$ the average number of subjobs per job at time t. If the cluster is not idle, the job splitting algorithm ensures that all nodes have a running subjob. Thus :

$$\forall t > 0 \quad j(t)\ s(t) \simeq n \tag{4.25}$$

In the case of the out of order scheduling policy, splitting of subjobs is carried out by ensuring that the maximum number of subjobs can start immediately after their creation. Let us assume that all subjobs are always running. Thus, the average speedup at time t is proportional to $s(t)$. Provided that the cluster is running $j(t)$ jobs concurrently and that the maximum average speedup is bounded (see Section 3.4.4), we have a ratio of subjobs and jobs which does not depend on the number of nodes :

$$\frac{s(t)}{j(t)} = O(1) \tag{4.26}$$

Therefore :

$$\forall t > 0 \qquad j(t) = O(\sqrt{n}) \tag{4.27}$$

$$s(t) = O(\sqrt{n}) \tag{4.28}$$

We deduce that the number of subjobs per job at a given time as well as the number of running jobs are both proportional to the square root of the number of nodes. Figure 4.6 gives a comparison of this result with the simulations.

4.4 Delayed Scheduling Model

4.4.1 Two phases

In the delayed scheduling policy, scheduling is only carried out once per period of time. Let H be the length of this period. If we assume that the system is not overloaded, we can consider that all jobs of the previous period have been terminated at the beginning of a new period. Thus processing operations carried out during the new period are divided in two parts :

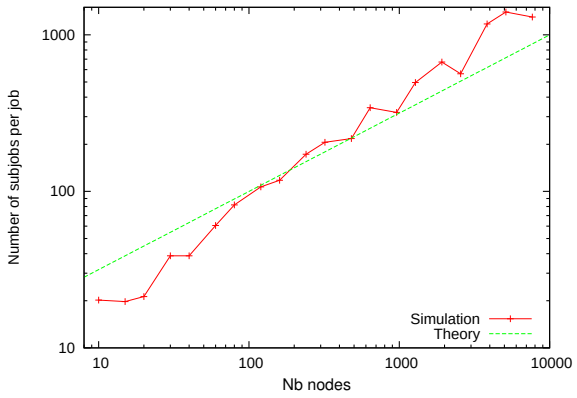

Figure 4.6: Average number of subjobs per job for the out of order scheduling policy compared with the theoretical model

- the processing of events that are present in caches
- the processing of all other events, after loading the data segments from tertiary storage.

Let T_1 and T_2 be the durations of these two phases and c be the portion of data cached on a single node. The number of jobs queued in front of a node after scheduling because they access cached data is :

$$\mathcal{N}_{cached} = \lambda \; H \; c \tag{4.29}$$

Let γ be the speedup when processing an event located in cache. Then

$$\begin{aligned} T_1 &= \frac{\mathcal{N}_{cached}}{\gamma \; \mu} \\ &= \frac{\lambda \; H \; c}{\gamma \; \mu} \end{aligned}$$

With $\rho = \frac{\lambda}{n \; \mu}$, we have :

$$T_1 = \frac{n \; c}{\gamma} \; \rho \; H \tag{4.30}$$

Let d be the portion of data (cached + non cached) read on average by a job. The average number of jobs that will access the data item associated to a given event during a period will be :

$$\mathcal{N}_{jobs} = \lambda \; H \; d = n \; \mu \; \rho \; H \; d \tag{4.31}$$

In the second phase of the processing, the data segments have to be loaded but they are only loaded once. The average speedup Γ of the jobs during this period can be estimated by considering that the speedup is 1 once every $\mathcal{N}_{jobs}$ jobs when the data has to be loaded and γ for all the others jobs, when the data is already loaded. This leads to :

$$\begin{aligned}
\Gamma &= \begin{cases} 1 & \text{if } \mathcal{N}_{jobs} \leq 1, \\ \frac{1}{\mathcal{N}_{jobs}}\, 1 + \frac{\left(\mathcal{N}_{jobs}-1\right)}{\mathcal{N}_{jobs}}\, \gamma & \text{if } \mathcal{N}_{jobs} \geq 1 \end{cases} \\
\Gamma &= \begin{cases} 1 & \text{if } \mathcal{N}_{jobs} \leq 1, \\ \gamma + \frac{(1-\gamma)}{n\ \mu\ \rho\ H\ d} & \text{if } \mathcal{N}_{jobs} \geq 1 \end{cases}
\end{aligned} \tag{4.32}$$

The total number of jobs to be processed in the second part is :

$$\begin{aligned}
\mathcal{N}_{non\ cached} &= \lambda\, H - n\, \mathcal{N}_{cached} \\
&= \lambda\, H\, (1 - n\, c)
\end{aligned} \tag{4.33}$$

Since these jobs will be processed in parallel on all nodes of the cluster, we deduce :

$$\begin{aligned}
T_2 &= \frac{\mathcal{N}_{non\ cached}}{n\, \mu\, \Gamma} \\
&= \frac{\lambda\, H\, (1 - n\, c)}{n\, \mu\, \Gamma} \\
T_2 &= \frac{\rho\, H\, (1 - n\, c)}{\Gamma}
\end{aligned} \tag{4.34}$$

4.4.2 Average waiting time

Two cases have to be considered depending whether a job has some of its data cached or not. In case a job has cached data, it will start during phase one. If we neglect the processing time of a single job compared to the duration T_1 of the first phase, we can say that on average, such a job has a waiting time of $\frac{T_1}{2}$.

When a job has no data cached at all, it will start during phase two, i.e. it will wait during the whole first phase plus, on average, half of the second phase (we neglect the processing time of a single job compared to the duration T_2 of the second phase). Its waiting time will thus be $T_1 + \frac{T_2}{2}$.

Let us call α the probability that a job has some of its data cached. We get the average waiting time per job AWT :

$$\begin{aligned} AWT &= \alpha \frac{T_1}{2} + (1-\alpha)\left(T_1 + \frac{T_2}{2}\right) \\ &= T_1\left(1 - \frac{\alpha}{2}\right) + T_2 \frac{1-\alpha}{2} \\ AWT &= \rho\, H \left[\frac{n\, c}{\gamma}\left(1 - \frac{\alpha}{2}\right) + \frac{1 - n\, c}{\Gamma}\left(\frac{1-\alpha}{2}\right)\right] \end{aligned} \tag{4.35}$$

where only Γ is not a constant (at least when $\mathcal{N}_{jobs} \geq 1$).

It is difficult to derive a precise number for α since it depends on several difficult to establish parameters such as the cache policy, the stripe size and the job size distribution. However, we can have a rough estimate of the average waiting time by using $\alpha = n\, c$, i.e. the portion of the data space contained in the caches. This lead to :

$$AWT = \rho\, H \left[\frac{n^2\, c^2}{2\, \gamma} + \frac{n\, c\, (1 - n\, c)}{\gamma} + \frac{(1 - n\, c)^2}{2\, \Gamma}\right] \tag{4.36}$$

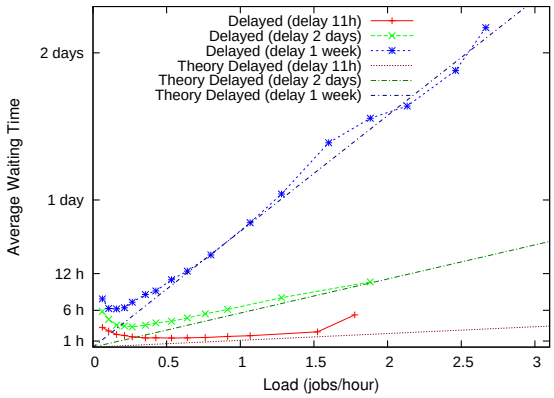

Figure 4.7: Average waiting time for delayed scheduling policy compared with equations

Figure 4.7 plots a comparison between the simulated average waiting time and formula 4.36. It shows that the approximation is valid in the range of loads between 10% and 90% of the maximal load.

4.4.3 Average speedup

While we had to consider two cases for the waiting time, we have to consider three cases for the derivation of the average speedup :

- jobs with only cached data
- jobs with no cached data
- jobs with partially cached data and partially non cached data

In the first two cases, the jobs start and end within the same phase while in the third case, the jobs start in the first phase and then stop and wait for the second phase to resume and complete their execution.

If a job has only cached data, its average service time AST can be deduced from the service time on a single node with no cache (i.e. $\frac{1}{\mu}$) by taking into account the improvement due to the cache and the improvement due to the parallelization. The improvement due to the cache is γ. We call $d_{//c}$ the improvement due to the parallelization. We obtain :

$$AST_{cached} = \frac{1}{\gamma \, \mu \, d_{//c}} \tag{4.37}$$

where $d_{//c}$ can be seen as the average degree of parallelization of the jobs with only cached data, i.e. the average number of nodes on which they run. Note that due to the specificities of the delayed scheduling policy (ignorance of the job arrival order) and to the random distribution of the data used by the different jobs, $d_{//c}$ does not depend on the load nor on the delay. Thus, it is a constant in our context.

The same reasoning line applies to jobs without cached data, leading to the same kind of formula for their average service time :

$$AST_{non\ cached} = \frac{1}{\Gamma \, \mu \, d_{//nc}} \tag{4.38}$$

where $d_{//nc}$ can be seen as the average degree of parallelization of the job without cached data.

In the case of jobs with both cached and non cached data, we have to take into account their waiting phase. For typical loads and delays, this waiting phase is far longer than the job processing time. We thus neglect the processing time and

suppose that the average service time is equal to the duration of the waiting phase, i.e. on average :

$$\begin{aligned} AST_{mixed} &= \frac{T_1 + T_2}{2} \\ AST_{mixed} &= \frac{\rho\, H}{2}\left(\frac{n\, c}{\gamma} + \frac{1 - n\, c}{\Gamma}\right) \end{aligned} \tag{4.39}$$

Let finally α and β be the probabilities that a job has only cached data and respectively no cached data. Then we have :

$$\begin{aligned} AST &= \alpha\, AST_{cached} + \beta\, AST_{non\ cached} + (1 - \alpha - \beta)\, AST_{mixed} \\ &= \left(\frac{\alpha}{\gamma\, d_{//c}} + \frac{\beta}{\Gamma\, d_{//nc}}\right)\frac{1}{\mu} + \frac{1 - \alpha - \beta}{2}\,\rho\, H\,\left(\frac{n\, c}{\gamma} + \frac{1 - n\, c}{\Gamma}\right) \end{aligned} \tag{4.40}$$

And by definition of the speedup :

$$\begin{aligned} Speedup &\triangleq \frac{1}{\mu\, AST} \\ Speedup &= \frac{1}{\left(\frac{\alpha}{\gamma\, d_{//c}} + \frac{\beta}{\Gamma\, d_{//nc}}\right) + \frac{1-\alpha-\beta}{2}\,\left(\frac{n\, c}{\gamma} + \frac{1-n\, c}{\Gamma}\right)\,\rho\, H\, \mu} \end{aligned} \tag{4.41}$$

Deriving theoretical estimates of the unknown values α, β, $d_{//c}$ and $d_{//nc}$ is beyond our knowledge. However, we can conclude about the form of the speedup and its dependency with respect to the load and the delay :

$$Speedup = \frac{1}{a + b\,\rho\, H\, \mu} \tag{4.42}$$

where a and b are considered as constants which do not depend on the load.

Figure 4.8 shows how expression 4.42 fits the simulation data for different delays and stripe sizes. The values of a and b are fitted from the simulated data and given in Table 4.1. Note that the actual values of α, β, $d_{//c}$ and $d_{//nc}$ may be extracted from the simulation data. This would require more simulations where the cache setup and the value of the gain γ are changed in addition to the load and the delay. It is only by using these 4 degrees of liberty that all 4 parameters can be isolated.

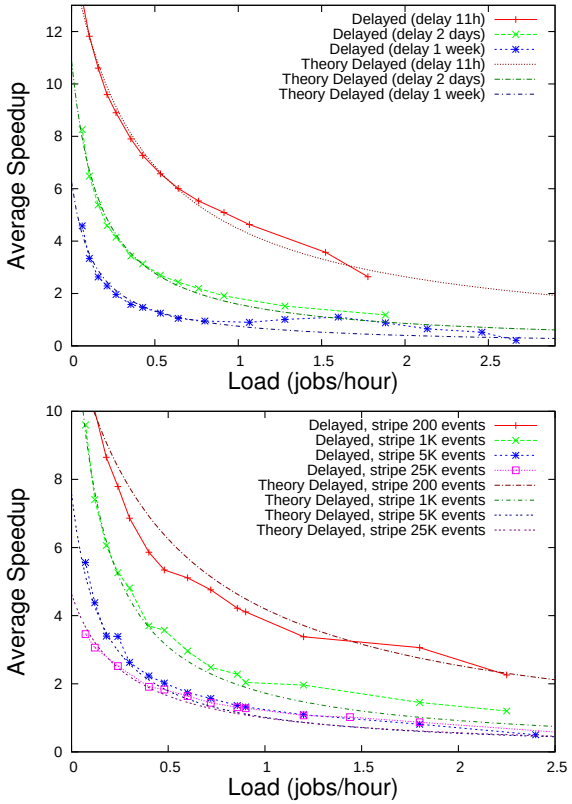

Figure 4.8: Average speedup for delayed scheduling policy compared with equations

4.5 Scheduling time complexities

The theoretical time complexity of each of the presented scheduling algorithms can be derived by a careful analysis of their implementation. Table 4.3 presents the main lines of this analysis leading to the scheduling time complexities per job summarised Table 4.2. The complexities are given with respect to the number n of computing and storage nodes present in the system.

Table 4.2 gives the complexity of scheduling one job depending on the number n of nodes in the cluster. However, in a real case, the number of jobs processed per unit of time is also dependent on the number of nodes, typically proportional to it. Taking this fact into account, theoretical complexities can be compared with the measurements from the simulation system. Figure 4.9 gives simulation for the following case :

delay	stripe size (nb events)	a	b
11 h	1000	.062	.248
2 days	1000	.070	.163
1 week	1000	.072	.090
2 days	200	.056	.074
2 days	1000	.070	.163
2 days	5000	.132	.273
2 days	25000	.216	.252

Table 4.1: Coefficients used for fitting delayed scheduling data with theoretical equations

Policy	Complexity
Processing Farm	$O(n)$
Job Splitting	$O(n^2)$
Out of order	$O(n\sqrt{n})$
Delayed Scheduling	$O(n)$

Table 4.2: Scheduling time complexities of the different scheduling algorithms presented in respect to the number n of nodes in the cluster

1. The cluster load is proportional to the number of nodes

2. The data space is size proportional to the number of nodes

3. For delayed scheduling, the period delay is inversely proportional to the number of slave nodes. Thus the average number of jobs arriving into the cluster within a period is not dependent on the number of slave nodes. The delay used in Figure 4.9 is one month for a one node cluster, corresponding to e.g. 3 days for a 10 nodes cluster and 7 hours for a hundred nodes cluster.

Figure 4.9 shows both the simulation data (marker lines) and the theoretical models (dotted, without marker lines). Only the slope of the lines were known and their position was derived from the simulation data.

Simulations and theory match well for the job splitting and out of order policies. In the case of the delayed policy, the load represents both the subdivision of data into the desired stripe size and job splitting. Thus an extra curve of simulated data shows the time spent only in the job splitting part of the delayed scheduling

Table 4.3: Details on the computation of the theoretical scheduling time complexities of the different algorithms presented with respect to the number n of nodes in the cluster.

Processing Farm	Out Of order
Scheduling one job requires traversing the list of nodes to find an idle node. Scheduling complexity is $O(n)$.	Upon job arrival, the new job has to be split into subjobs. Splitting takes caches into account by reading the content description of each node cache, yielding a complexity of $O(n)$.
Job Splitting	When a node becomes available, a subjob needs to be split. By looking at all of them (one per node), the complexity is $O(n)$ per split. The global complexity per job is thus of the order of $O(n)$ multiplied with the average number of splits per job, i.e. the average number of subjobs. According to Section 4.3, the average number of subjobs is $\sqrt{n}$. the complexity per job is thus $O(n\sqrt{n})$.
Upon job arrival, the worst case requires scanning all nodes to find the subjob to be suspended. Complexity of this case is $O(n)$.	
Upon job or subjob end, the worst case consists in finding the subjob to split. By looking at all of them (one per node) the complexity is $O(n)$ per split.	
According to the theoretical model of Section 4.2, the average number of jobs in the system depends only on the load, and is independent of the number of nodes n. Since the jobs are parallelized across all nodes, the number of splits per job is $O(n)$. Since the number of splits per job is $O(n)$ and each split operation is $O(n)$, the overall complexity of single job splitting is $O(n^2)$.	**Delayed Scheduling**
	At each new period, the new jobs are split into subjobs. Job splitting takes caches into account by reading the content description of each node cache, yielding a complexity of $O(n)$.
	The non cached subjobs are further split. The number of splits per job is given by the stripe size and is independent of the number of nodes yielding a complexity of $O(1)$. The overall complexity per job is thus $O(n)$.

algorithm. It clearly shows that the complexity of the job splitting part matches the theoretical complexity. However, the simulations show that the job splitting part becomes preponderant only for a large number of nodes, around 50 000.

From the extrapolations of the simulated data via the theory, the expected maximum number of nodes that a single master node can handle is around 5 000 for

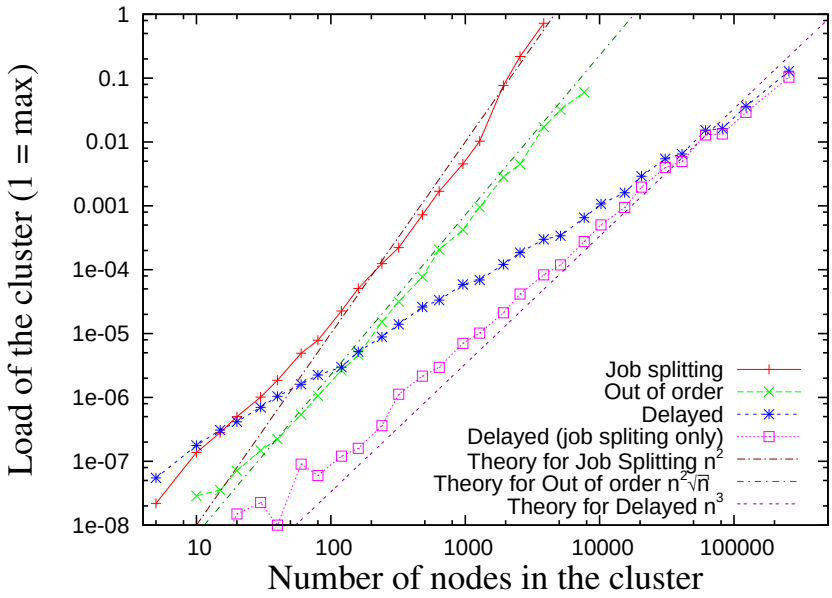

Figure 4.9: Average scheduling load of the master node as a fraction of the maximum load in function of the number of nodes in the cluster. The tests run on a single 2.0GHz Intel Celeron processor, the delay for delayed scheduling is one month divided by the number of nodes

the job splitting policy, around 20 000 for the out of order scheduling policy and around 300 000 for the delayed scheduling policy. The period delay introduced by the delayed scheduling policy reduces the job scheduling load for very large clusters by an order of magnitude.

4.6 Scheduling Memory Space complexity

The memory space complexity is bound by the amount of data needed by the master node to schedule the jobs, namely the list of nodes and the status of their disk caches. The required memory space is linear with the number of nodes and remains relatively small :

- The list of nodes is not large compared with the representation of the information present in the disk caches
- The size of the representation of the information contained in a disk cache is in the order of the cache size divided by the stripe size. The maximal

cache size being around a few terabytes and the stripe size comprising a few events i.e. a few megabytes, the size of the representation of cached information is a few megabytes.

We may therefore store the information contained in two thousands very large disk caches (each disk 500 GB yielding a total size of 1 PB) divided into very small stripes within 1GB of RAM. For a regular stripe size, the scalability limitation due to memory limitations on the master node is of the order of tens of thousands of slave nodes in the cluster.

4.7 Conclusion

We develop theoretical model for some of the scheduling algorithms elaborated in chapter 3. The job splitting scheduling policy model is derived from the theory of queueing systems and matches very well the simulation data.

The out of order scheduling policy is approximated by extending equations of the job splitting scheduling policy model. The resulting model is not very precise (see Figure 4.4) since it does not take into account the effects of subjobs reordering on the jobs themselves. Improved equations (4.22 to 4.24) are derived from the simulation data but more research effort are needed to derive them directly from the theory.

The delayed scheduling policy model gives an exact formula for the average waiting time (see equation 4.36) but only the shape of the speedup. Some parameters are actually unknown and have to be deduced from the simulation data. Comparison of theory and simulation data shows a very good match (Figures 4.7 and 4.8).

We finally study the scheduling time and space complexities of the different scheduling algorithms and show that a single master node can handle from 5000 to 200 000 slave nodes, depending on the scheduling policy (Figure 4.9).

Chapter 5

Pipelining processing and I/O operations

In the previous chapters, we considered that I/O operations are performed serially, i.e. processing operations can operate on data only after the disk accesses have been carried out. However modern computing systems allow to asynchronously perform I/O and processing operations and to hide the shortest of the two activities. We consider two types of pipelining :

- pipelining of tertiary storage access and computation : the access to tertiary storage and the processing are performed asynchronously
- pipelining of local disk I/O, tertiary storage access and computation : reading data segments from the disk caches is pipelined with the processing activities

5.1 Pipelining accesses from tertiary storage

The computation of a given event can be carried out in parallel with the loading of the next event from the tertiary storage (if the next event is not cached). With this strategy, we expect to be able to hide completely the computation activity for non cached events. The theoretical speedup for an event read from tertiary storage is $\frac{T_{CPU}+T_{I/O}}{T_{I/O}} = \frac{0.2+0.6}{0.6} = 1.33$, i.e. a performance improvement of 33%.

5.1.1 First come, first served scheduling policies

Figure 5.1 shows the improvement in speedup due to pipelining according to simulations for both the processing farm and the job splitting policy. The theoretical value of 33% is confirmed for the processing farm policy : the speedup is stable at 1.33 with pipelining.

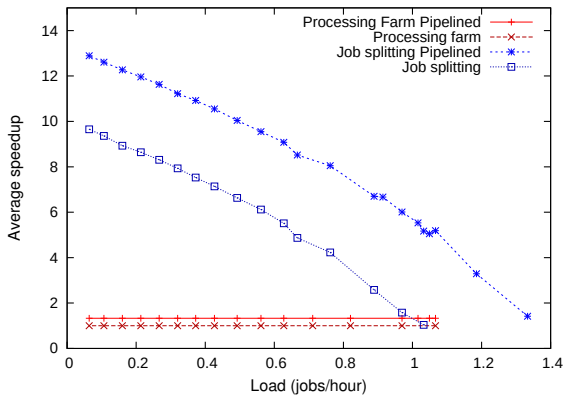

Figure 5.1: Average speedup for different scheduling policies with and without pipelining of tertiary storage accesses, deduced from simulations.

For the job splitting policy, the theoretical speedup improvement of 33% is confirmed at low loads. We obtain speedup of 13 with pipelining and of 9.75 without pipelining. At higher loads, the speedup improvement is higher than the theoretical value. This is mainly due to a virtuous cycle starting with the reduction of the average time spent by the jobs in the cluster. Whenever a job spends less time in the cluster, it leaves its nodes earlier. The other running jobs can take advantage of these nodes to increase their parallelization degree and run faster. They will in turn stop earlier, leaving even more nodes for the next jobs.

Applying the theoretical results of Section 4.2 and letting Λ be the average improvement for a single job on a single node (here $\Lambda = 1.33$), the average service time AST_{TP} with pipelining of tertiary storage access is :

$$AST_{TP} \approx \frac{1}{n\ \mu\ \Lambda\ (1 - \frac{\rho}{\Lambda})} \tag{5.1}$$

where the mean service time μ of equation (4.16) has been replaced by $\frac{\mu}{\Lambda}$ for the pipelined case. Note that since the load $\rho = \frac{\lambda}{\mu}$ and ρ is being divided by Λ, we

obtain the speedup :

$$Speedup_{TP} \approx n\,\Lambda\,(1-\frac{\rho}{\Lambda}) \quad (5.2)$$

$$\approx n\,(\Lambda-\rho) \quad (5.3)$$

The speedup improvement of a factor Λ at low loads is confirmed (See Figure 5.2 for load 0.05). For higher loads, the speedup of the job splitting strategy decreases linearly with the load along the same slope as in the case with no pipeline. The theory and the simulations match very well as shown by Figure 5.2. Figure 5.1 shows that the speedup improvement of 33% at low loads becomes 50% for 0.5 jobs per hour, 100% for 0.9 jobs per hour and 300% for 1 jobs per hour. The maximum sustainable load of the cluster using the job splitting policy increases from 1.15 to 1.5 jobs per hour, i.e. by a factor Λ as predicted by the theory.

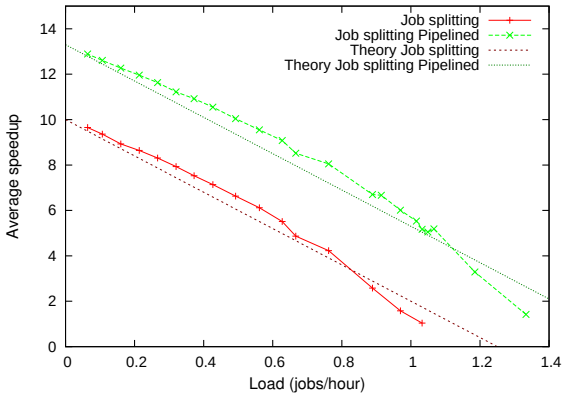

Figure 5.2: Average speedup for the job splitting scheduling policy with pipelining of tertiary storage accesses compared with the theoretical estimations

5.1.2 Out of order scheduling policy

Figure 5.3 shows the gain of pipelining for the out of order scheduling policy, according to the simulations. The relative speedup gain is proportional to the amount of data that is loaded from tertiary storage. This leads to large improvements in case of a small disk cache and no improvement at all when all data is located within the disk caches. The maximum load supported by the cluster is also improved in the case of small disk caches. From a performance point of view,

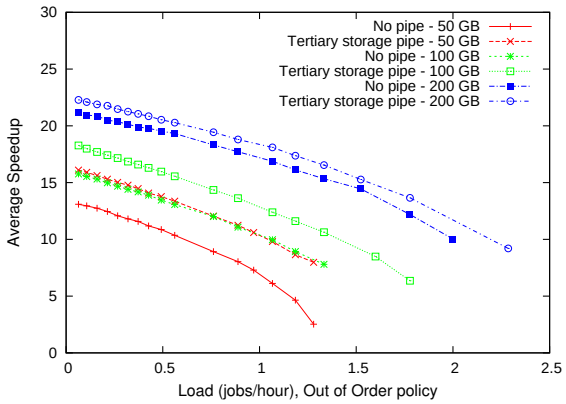

Figure 5.3: Average speedup for the out of order scheduling policy with and without pipelining of tertiary storage accesses for different disk cache sizes and different load levels, deduced from simulations

pipelining has the same effect on the out of order scheduling policy as increasing the disk cache size.

Applying the theoretical results of Section 4.3 and letting Λ be the average improvement for a single job on a single node (here $\Lambda = 1.33$), the average speedup Γ_{TP} with pipelining of tertiary storage access is :

$$\Gamma_{TP} = \Lambda + n\, c\, (\gamma - \Lambda) \tag{5.4}$$

Using equations 4.20 and 4.21, we can then deduce the average service time and speedup with pipelining of tertiary storage access :

$$AST \approx \frac{1}{n\, \mu\, \Gamma_{TP} \left(1 - \frac{\rho}{\Gamma_{TP}}\right)} \tag{5.5}$$

$$Speedup \approx n\, (\Gamma_{TP} - \rho) \tag{5.6}$$

Figure 5.4 shows a comparison of the theoretical computations and the simulations results. As in the case without pipelining, the theoretical computation is not precise enough although the order of magnitude of the speedup is correct.

However, reusing the improved equations given in Section 4.3.2 we can write :

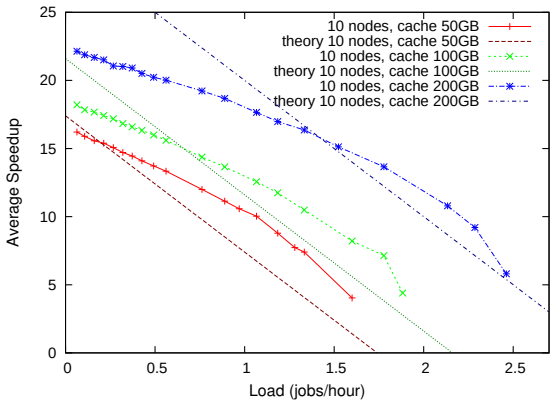

Figure 5.4: Average speedup for the out of order scheduling policy with pipelining of tertiary storage accesses compared with theoretical computations

$$\Gamma_{TPi} = \Lambda + \frac{3}{4}\, n\, c\, (\gamma - \Lambda) \tag{5.7}$$

$$AST_{TPi} \approx \frac{1}{n\, \mu\, \Gamma_{TPi} \sqrt{1 - \frac{\rho}{\Gamma_{TPi}}}} \tag{5.8}$$

$$Speedup_{TPi} \approx n \sqrt{\Gamma_{TPi}} \sqrt{\Gamma_{TPi} - \rho} \tag{5.9}$$

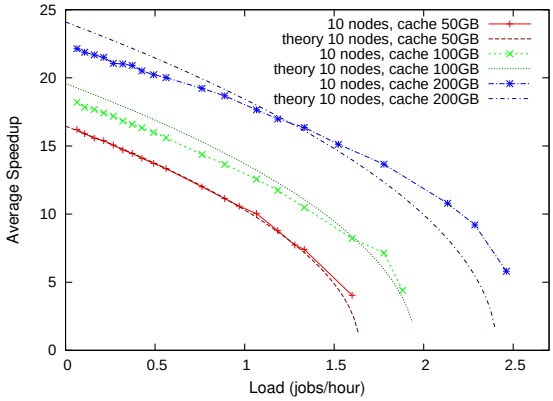

Figure 5.5: Average speedup time for out of order scheduling policy with pipelining of tertiary storage accesses compared with the theoretical model

Figure 5.5 shows that the improved equations match accurately the simulations.

5.1.3 Delayed scheduling policy

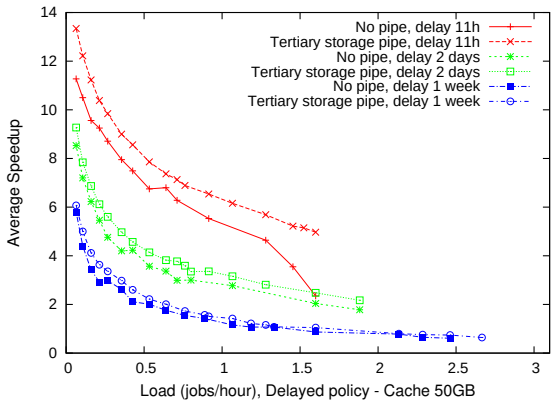

Figure 5.6: Average speedup for the delayed scheduling policy with and without pipelining of tertiary storage accesses for a small disk cache and different load levels

Figure 5.6 shows the gain of pipelining for the delayed scheduling policy according to the simulations. The disk cache size was fixed to 50GB and different delays were simulated. The improvements are also proportional to the proportion of data read from tertiary storage. Since the delayed scheduling policy makes a heavy use of disk caches, improvements are only present for small delays. Note that the same simulations with a disk cache of 100GB show no improvement at all, even for small delays, confirming that access to tertiary storage is negligible for this cache size.

5.2 Pipelining of cache disk I/O

Pipelining of cache disk I/O and event processing allows us to further reduce the overall processing time by hiding the disk I/O in the case of cached data. When pipelining disk I/O, the theoretical speedup for an event read from disk is $\frac{T_{CPU}+T_{I/O}}{T_{CPU}} = \frac{0.2+0.06}{0.2} = 1.3$. Pipelining of cache disk I/O brings no speedup when the event is loaded from tertiary storage. However, in such a case, pipelining from tertiary storage access brings an important speedup. The two pipelining capabilities are therefore complementary.

Note that disk I/O pipelining makes no sense in the case of the computing farm and job splitting policies since, for these policies, data segments are always read

from tertiary storage.

5.2.1 Out of order scheduling policy

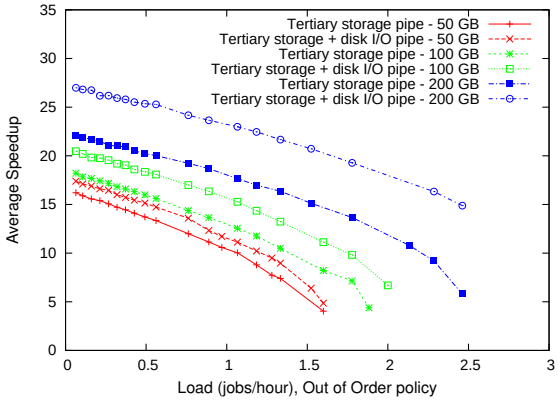

Figure 5.7: Average speedup for the out of order scheduling policy with tertiary storage pipelining only and with both tertiary storage and disk I/O pipelining for different disk cache sizes and different load levels

Figure 5.7 shows the gain of pipelining disk I/O and tertiary storage access in the case of the out of order policy compared to the case where only the access to tertiary storage is pipelined. The simulations show that the gain is large for large disk caches where most of the data is read from the disk caches while it is small for small disk caches where most of data is read from tertiary storage. Since the performance gain of disk I/O pipelining is complementary to the gain obtained by tertiary storage pipelining, the total gain in speedup due to both disk I/O and tertiary storage pipelining is quite constant, around 30% as shown in Figure 5.8.

Reusing Section 4.3, we can establish the theoretical model for the case where both disk I/O and access to tertiary storage are pipelined. Let η be the speedup in case of disk I/O pipelining for a cached event (here we have $\eta = 1.3$) and Γ_{TDP_i} the average speedup where $_{TDP_i}$ means respectively "Tape", "Disk", "Pipelining" and "improved". We adapt Equation 5.7 deduced from the simulations and obtain :

$$\Gamma_{TDP_i} = \Lambda + \frac{3}{4}\, n\, c\, (\eta\, \gamma - \Lambda) \tag{5.10}$$

According to Equations 5.8 and 5.9, we derive the following equations for the service time and the speedup :

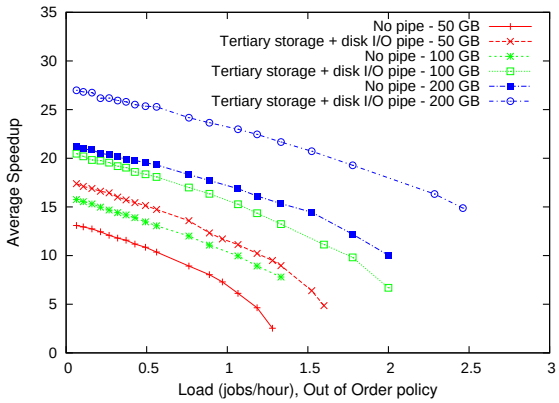

Figure 5.8: Average speedup for the out of order scheduling policy with no pipelining and with both tertiary storage and disk I/O pipelining for different disk cache sizes and different load levels

$$AST_{TDP_i} \approx \frac{1}{n\,\mu\,\sqrt{\Gamma_{TPi}}\sqrt{\Gamma_{TDP_i}-\rho}} \tag{5.11}$$

$$Speedup_{TDP_i} \approx n\,\sqrt{\Gamma_{TPi}}\sqrt{\Gamma_{TDP_i}-\rho} \tag{5.12}$$

Note the use of both Γ_{TPi} and Γ_{TDP_i} in the equations. Figure 5.9 shows that equations 5.11 and 5.12 fit the simulation data.

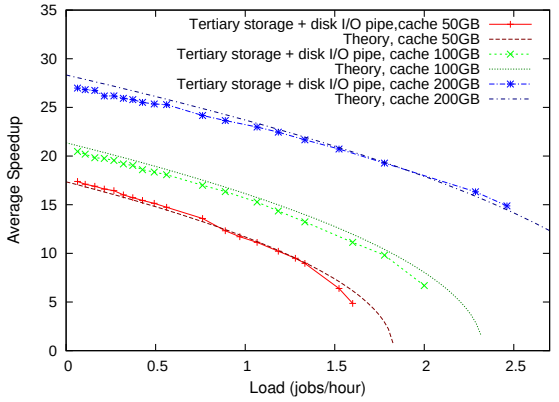

Figure 5.9: Average speedup for the out of order scheduling policy with both tertiary storage and disk I/O pipelining compared with the theoretical model

5.2.2 Delayed scheduling policy

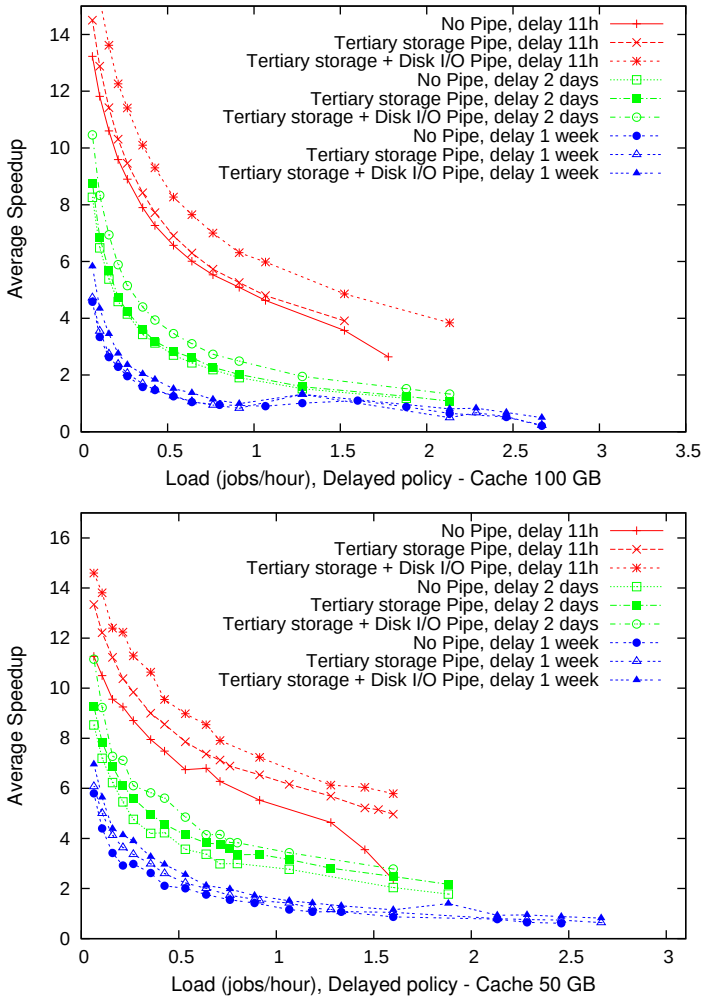

Figure 5.10: Average speedup for the delayed scheduling policy with tertiary storage pipelining only and with both tertiary storage and disk I/O pipelining for different disk cache sizes and different load levels

Figure 5.10 shows the gain of pipelining the disk I/O in the case of the delayed policy compared to the case where only the tertiary storage accesses were pipelined. The first figure shows simulations run with a disk cache of 100GB. In the second figure, the disk cache is only 50GB.

In the 100GB case, in a first approximation, tertiary storage accesses are neg-

ligible. We thus expect a speedup of 30% due to pipelining of disk I/O. This is actually confirmed for all loads when the delay is 11h or 2 days. In the case of long delays (1 week), the speedup improvement is hidden by the very long delays that the delayed scheduling policy imposes.

In the 50GB case, due to the smaller cache, the impact of disk I/O pipelining is smaller, especially for short delays, e.g. the improvement for a delay of 11h is only 15%.

5.3 Conclusions

Pipelining tertiary storage accesses and computations increases performances of scheduling policies when caches are not heavily used. The gain in speedup increases up to 30% in cases where the cache is not used.

Pipelining disk I/O and computation increases the performances of scheduling policies where the cache is heavily used. The gain in speedup increases up to 30% and it adds to the gain obtained the pipelining of tertiary storage accesses.

Using both pipelining techniques, an average speedup gain of 30% is observed in case of job splitting and out of order scheduling policies. However, the speedup improvement is smaller for the delayed scheduling policy due to the long delays it suffers from.

Theoretical models of pipelining are based on the modelization of the different scheduling policies. As a consequence, the model of pipelining for the job splitting policy matches very well the simulation results while the model of the out of order scheduling policy is not very accurate. As in Chapter 4, we give improved equations for the out of order scheduling policy, deduced from the simulation results. More work is needed to derive them directly from the theory.

Chapter 6

Integration in LHCb's framework

In the previous chapters, we have studied in details the different scheduling algorithms for parallelizing physics data analysis jobs on clusters of PCs.

We will now show how this parallelization can be implemented on a real particle physics application, namely the LHCb software. All the application of LHCb are based on a common framework called GAUDI [cg05]. Our goal is to implement the parallelization at the framework level so as to ensure that all applications can benefit from the parallelization with no modification of their own code.

We will first present the GAUDI framework and the DPS [GH03] parallelization tool before describing the implementation and the status of our prototype.

6.1 GAUDI, the LHCb framework

The GAUDI framework aims at providing a set of tools that facilitate the creation of particle physics applications. Section 6.1.1 presents the major design criteria of GAUDI. Section 6.1.2 presents an overview of the main components of GAUDI and Section 6.1.3 describes its event loop concept.

6.1.1 Major design criteria

Clear separation between data and code

Data handled by GAUDI applications are events, geometry data and histograms. The processing code is provided by physicists and is mainly contained in algorithms. Algorithms are stateless pieces of code and implement some data trans-

formation process. As an example, an algorithm can compute statistics and create histograms or reconstruct particle tracks and fill the fields of the event data items.

Data centered architecture

The architecture allows the development of physics algorithms in a fairly independent way. A transient data store is used for communication between algorithms. Data objects can be registered and identified through a hierarchical addressing scheme. The transient store is a pure memory object. Algorithms do not have to care about data persistency.

Clear separation between persistent data and transient data

Persistent data items reside on disk or tape while transient data items reside in memory, i.e. in transient stores. The conversion between transient and persistent data is carried out automatically by the *DataSvc* service and its sets of data converters. The conversion is carried out on demand by the framework.

Well defined interfaces, as generic as possible

Each component of the architecture implements a number of abstract interfaces used for interacting with the other components. Therefore, the implementation of a component can be replaced without changing the rest of the application.

6.1.2 Main components

A picture of the main components of GAUDI is given in Figure 6.1.

Algorithms and Application Manager

The essence of GAUDI applications is event processing, i.e. computations based on particle physics events. The processing code is encapsulated into a set of components called algorithms. Algorithms implement an abstract interface that allow to call them without knowing what they really do. By calling each other, algorithms build a hierarchical structure at the top of which the application manager sits. The application manager is the "chef d'orchestre" who decides what algorithms to load and when to call them, based on the application's configuration file.

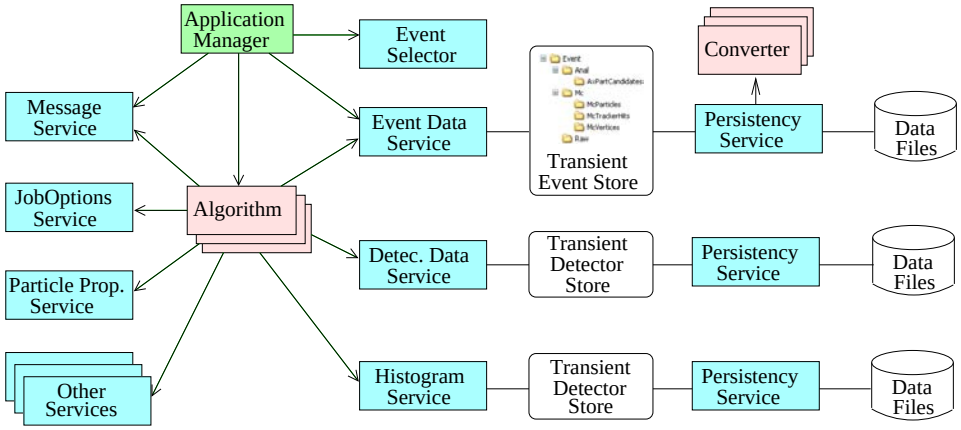

Figure 6.1: GAUDI architecture : Component view

Transient data stores

The data objects needed by the algorithms are organised in various transient data stores : event data store, detector data store and histogram data store. These are pure memory object stores, automatically populated on demand from event, geometry or histogram files by dedicated services using appropriate sets of data converters.

Services

Services are components which offer useful functionalities to physicists, enabling them to concentrate on developing the physics content of the algorithms. Examples of services can be seen in the object diagram on Figure 6.1. For instance, there are services for managing the different transient stores (event, detector and histogram data service), sending messages to the user, dealing with options, accessing a repository of particle properties, ...

Event Selector

The Event Selector provides functionality to the application manager for selecting the events that the application should process. This will be one of the main targets for our parallelization efforts, together with the Application Manager itself.

6.1.3 The event loop

The Event Loop is the central part of the GAUDI architecture. It is handled by the Application Manager and is supposed to run a given list of algorithms on each event of the input set. Figure 6.2 gives a schema of a Gaudi application from the event loop point of view.

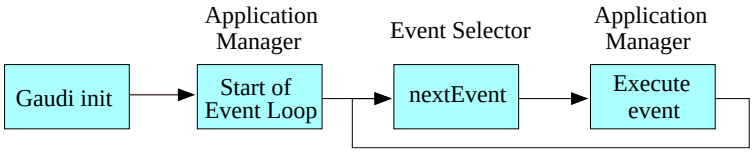

Figure 6.2: The Event Loop

Inside the *Execute Event* operation, algorithms are run in a specific order, given by the configuration file of the GAUDI application. Everything runs as a data flow from one algorithm to the next, as shown on the left part of Figure 6.3.

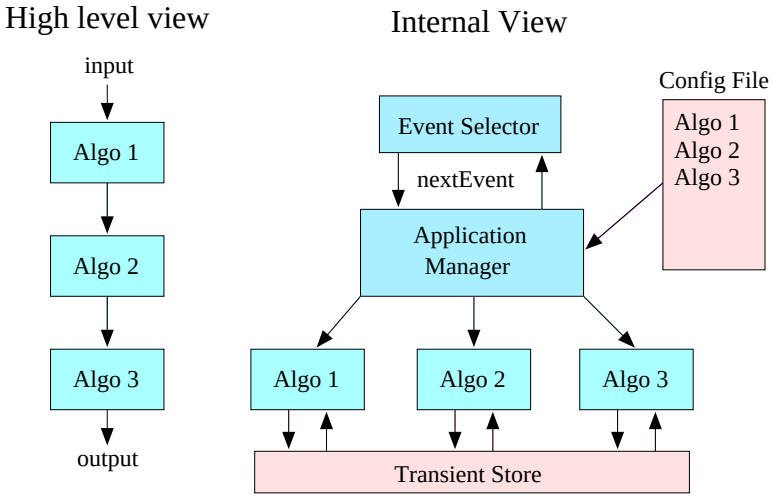

Figure 6.3: GAUDI high level and internal view

The actual execution is shown on the right part of Figure 6.3. The application manager uses the event selector to run through the events and runs the algorithms one by one as specified by the configuration file. From its point of view, algorithms are completely independent. The transmission of the data from one algorithm to the next is carried out through the transient store.

6.2 DPS, the dynamic parallelization system

DPS (Dynamic Parallel Schedules) [GH03] is a framework for developing distributed parallel applications on clusters of PCs or workstations. DPS uses a high-level graph-based application description model, allowing for complex application designs. DPS also provides a dynamic application execution environment, allowing for malleable applications and fault-tolerance.

6.2.1 Split - Merge construct

An application using DPS is expressed as a directed acyclic graph of sequential operations, called a flow graph. The flow graph is built of compositional customisable split - process - merge constructs as shown in Figure 6.4.

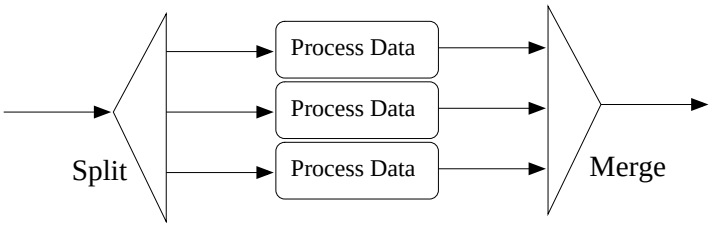

Figure 6.4: Unfolded split-operation-merge flow graph

The Split operation takes as input one data object, and generates as output multiple output data objects representing the subtasks to execute. Each Process-Data leaf operation processes the incoming data object and generates one output data object. The Merge operation collects all the output data objects to produce one global result. The programmer does not have to know how many data objects arrive at the merge operation, since DPS keeps track of the number of data objects generated by the corresponding split operation.

6.2.2 Flow graph representation

An application using DPS is described by its flow graph, also called parallel schedule. This description contains a sequence of operations passing data objects from one operation to the next operation.

The graph elements are compositional: a split-merge construct may contain another split-merge construct. A split-merge construct may incorporate different

paths, enabling a data-dependent conditional execution of parts of the flow graph at run time. Leaf operations can be sequenced, thus enabling the pipelined execution of operations in the flow graph. This allows overlapping computations, communications and I/O operations.

As it stands, a flow graph only indicates the processing operations and their order of execution. To enable parallelism, the operations need to be mapped onto different threads, possibly located on different processing nodes. Threads are mapped onto the nodes where their operations will execute. The threads are grouped into thread collections. A user-defined routing function specifies at run time in which instance of the thread in the thread collection the operation will execute.

DPS flow graphs and the mapping of thread collections to operating system threads are specified dynamically at run time by the application.

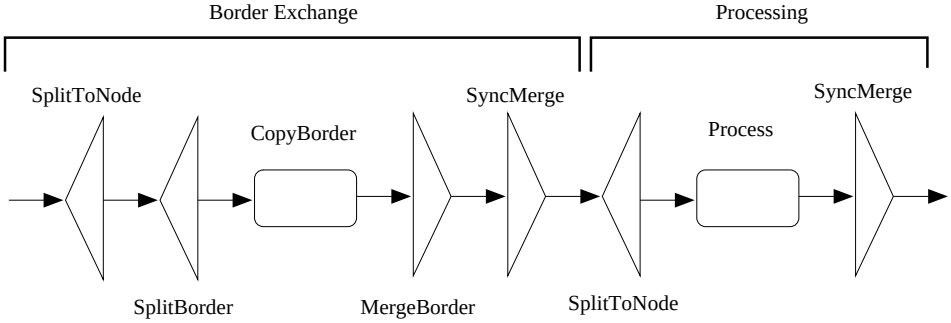

Figure 6.5: Flow graph for a game of life application

Figure 6.5 gives an example of a real life flow graph illustrating the parallelization of a neighbourhood dependant computation such as the game of life.

To parallelize the neighbourhood dependant computation (game of life), the world is split into horizontal tiles. Each node also stores the bottom-most line of the tile above it as well as the top-most line of the tile below it.

Each parallel iteration is then composed of two steps which have to be executed by all tiles. First the borders are exchanged by having each node asking its two neighbouring nodes to send their top and bottom rows to the requesting nodes. After this border exchange, the new state of the cells in the tile is computed.

The left part of the flow graph in Figure 6.5 parallelizes the border exchange between the different nodes.

- The first split operation asks all nodes to start performing the border exchanges.

- The second split is launched by each node to request from its neighbouring nodes the corresponding top and bottom border. The exchanges of the different borders are carried out in parallel.

The right part of the flow graph parallelizes the computation of the next step among the different nodes Figure 6.6 shows an expanded view of the execution of such a graph on 3 processing nodes.

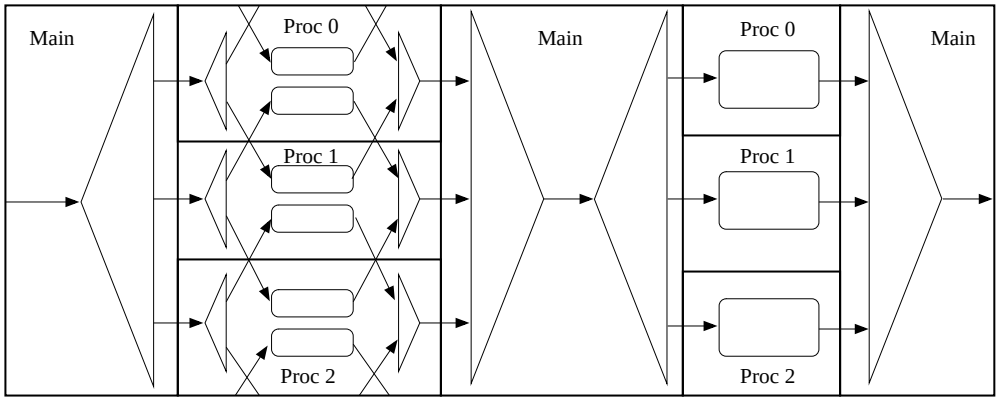

Figure 6.6: Expanded flow graph for a game of life application

6.3 Integration of GAUDI and DPS

6.3.1 Parallelizing the event loop

In order to parallelize the event loop of Figure 6.2, the loop over *nextEvent* and *Execute event* is replaced by a split merge structure. Each iteration of the split calls *nextEvent* and posts a *SplitToken*. The *Process Data* task calls *Execute event*. Figure 6.7 shows the equivalent flow graph.

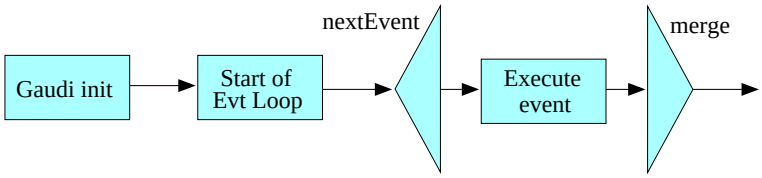

Figure 6.7: Parallelization of the GAUDI Event Loop

Figure 6.8 shows an unfolded view of the processing of the parallelized event loop on 2 threads distributed on 2 computer nodes. Node A is both the master node and a slave. Node B is only a slave. This figure describes how DPS executes the flow graph of Figure 6.7. Figure 6.8 should be compared to Figure 6.2. New items are painted in pink.

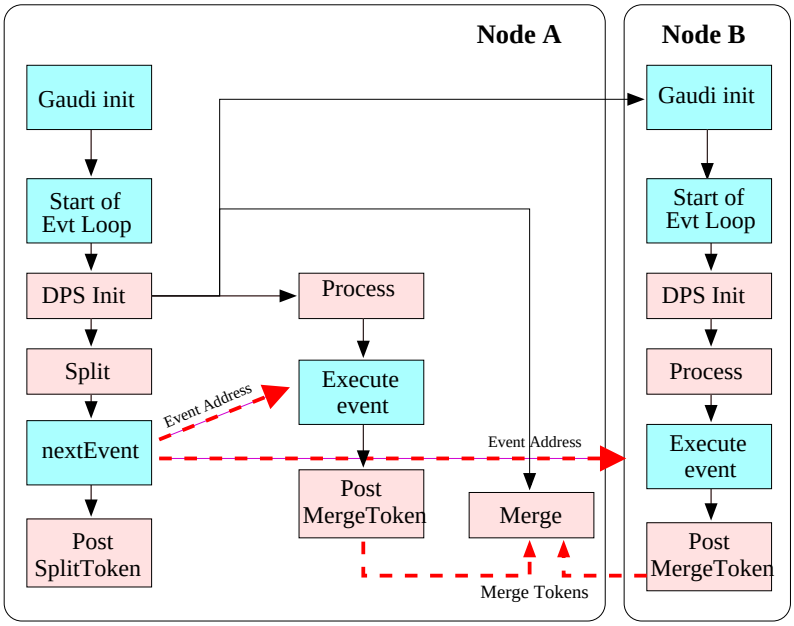

Figure 6.8: Integration of DPS in the GAUDI event loop

The sequence of actions is the following :

- GAUDI initialises and calls the event loop
- The DPS runtime is initialised. During initialisation, DPS go through the following steps :
  - it launches a new thread on the master node for the Merge part of the split-merge structure
  - it launches a new thread on node A for the Process part of the split-merge structure
  - it launches a GAUDI process on node B to run the Process part. This process runs the same application as the master node. It will also initialise GAUDI, which will start an event loop. This second event loop

will also initialise DPS and this second instance of DPS will communicate with the first one and launch a thread for the *Process* part

- GAUDI calls *nextEvent* in the Split thread. Its output is posted by DPS as a token to be used by one of the Process threads. The token contains the address of an event to process.

- One Process thread processes the event. This is where most of the GAUDI code runs. The output is packaged into a DPS token that is sent to the Merge thread using the DPS system.

- The Merge thread collects the results coming from all Process threads and produces the output files, as if everything had been produced by a single, standalone GAUDI process.

6.3.2 The LHCb DPS flow graph

The parallelization detailed in the previous section uses a single split-merge construct to parallelize the event processing on multiple nodes. However, as we have seen in Section 6.1, the algorithms could write the results into the transient stores. We thus need to collect the different stores on the different nodes and copy them to the store of the master.

This is carried out by using a second split-merge structure dedicated to the collection of store contents. The final flow graph is given in Figure 6.9.

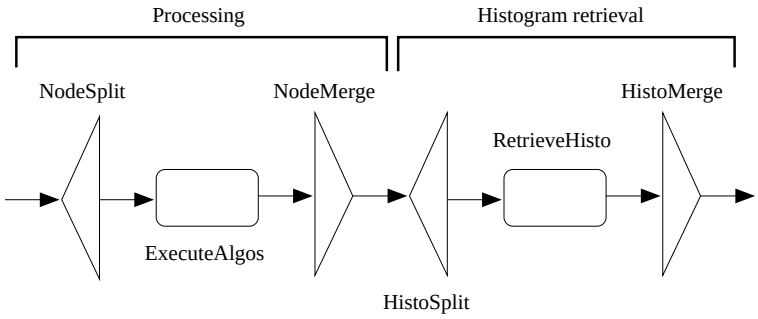

Figure 6.9: Flow graph of the parallel version of GAUDI

6.3.3 Implementing a parallel GAUDI

The architecture of the GAUDI framework allows to parallelize all applications based on GAUDI by only modifying two core components : the Application Manager and the Event Selector. These two components, as any other GAUDI component, have abstract interfaces that guaranty that no changes are needed in the application code when they are replaced by a different implementation. Furthermore, the choice of the implementation used can be done at run time by changing a single line in the configuration file of the GAUDI application.

As a proof of concept and feasibility, an implementation of a parallel version of the GAUDI Application Manager and Event Selector was written in 2003, using version v12r0 of the framework. This included the full flow graph presented in Figure 6.9. The provided event selector implemented the batch oriented scheduling policy.

This implementation was intended to be extended with the other scheduling policies described in the rest of this document. Although the policies were implemented in the simulation environment and can be reused as is in the GAUDI-DPS context, the resulting software could not be run due to clashes between DPS and a component introduced into newer versions of GAUDI : the ROOT [tea06] library. Both libraries are using threads in an incompatible way. After several weeks of debugging, it appeared that the ROOT software has to be modified in order to fix the situation which is not in the scope of the present thesis.

6.4 Conclusion

The ideas developed earlier in this thesis can be used to parallelize the different applications of the LHCb collaboration. All these applications are based on the same framework called GAUDI. Parallelizing the framework thus brings parallelizations to all the LHCb software.

GAUDI is designed with event processing in mind and controls the flow of incoming events in a dedicated service called the Event Loop. Using the Dynamic Parallel Schedule (DPS) system developed by EPFL, the Event Loop can be parallelized so that consecutive events are actually run in parallel on different machines. DPS allows a very comprehensive and elegant implementation of this parallel version of the event loop by separating the event processing itself from the parallelization code. The parallelization itself is handled by the DPS library, based on a description of the application under the form of a flow graph.

The integration of DPS and GAUDI was carried out successfully in an old version of the GAUDI. It was successfully tested on real data and provided parallelization to all LHCb application at the only cost of 2 additional lines in the configuration file. However, the parallel implementation of the event loop failed to be ported to newer versions of GAUDI due to incompatibilities between DPS and a library called ROOT. As a consequence, the scheduling policies presented in this work could not be tested on real data.

Note that even if the expected performances of the parallel version of GAUDI are identical to the simulations, it would anyway be difficult to compare the two approaches. Simulations are collecting statistics over several years of virtual running time. A realistic comparison with running programs will thus require studying how to accelerate the real case applications without modifying the scheduling statistics.

Conclusion

In this thesis, we have tried to solve several issues regarding the processing of particle physics data on clusters of PCs.

In chapter 2, we presented a new way of indexing and selecting data items in huge datasets having a high index dimensionality. The method avoids linear scanning of the whole data set. Only a minimal set of data is scanned, namely the values stored in tag collections. The selected tags point to the data items that are then retrieved for applying a more narrow selection.

By scanning tags in tag collections instead of a flat scan of all data items, the minimal gain is proportional to the ratio between the total number of data items and the number of tags within the selected tag collections. In many cases, the effective gain is the minimal gain multiplied by the ratio of the dimension of data items and the dimension of tags.

The proposed data items selection and retrieval scheme was implemented at CERN, in the context of the LHCb experiment and seems very promising. It was presented at the Conference on Mass Storage Systems and Technologies in April 2002 and published in the proceedings [PVH02].

In chapter 3, we propose several scheduling policies for parallelizing data intensive particle physics applications on clusters of PCs. We show that splitting jobs into subjobs improves the processing farm model by making use of intra-job parallelism. By caching data on the cluster node disks, cached-based job splitting further improves the performances.

The out of order job scheduling policy we introduce takes advantage of cache resident data segments and still includes a certain degree of fairness. It offers considerable improvements in terms of processing speedup, response time and sustainable loads. For the same level of performance, the typical load sustainable by the out of order job scheduling policy is double the load sustainable by a simple first in first out cache-based job splitting scheduling policy.

We proposed the concept of delayed scheduling, where the deliberate inclu-

sion of period delays further improves the disk cache access rate and therefore enables a better utilisation of the cluster. This strategy is very efficient in terms of the maximal sustainable load (50 to 100% increase) but behaves poorly in terms of response time and processing speedup. In order to offer a tradeoff between maximal sustainable load and response time, we introduce an adaptive delay scheduling policy with large delays at high loads and zero delays at normal loads. The delay is adapted to the current system load, thus trying to optimise the response time as a function of the current load. This adaptive delay scheduling policy aims at satisfying the end user whenever possible and at the same time allows to sustain high loads. All scheduling policies were presented at the 18th International Parallel and Distributed Processing Symposium in April 2004 and published in the proceedings [PH04].

In chapter 4, we built theoretical models for the different scheduling policies proposed in the previous chapters. The job splitting model is modelled by relying on the queueing theory. We present equations that match the simulations with a very good accuracy.

The out of order scheduling policy model was approximated based on the job splitting model. The approximated model does no closely match the simulations. We therefore propose more accurate equations that match perfectly the simulations but are variations of the results derived from the theoretical considerations.

For the delayed scheduling policy model, we proposed precise equations for the waiting time as well as the general form of the equations of the speedup. Although all the parameters of the equation could not be derived, we showed that a right choice of these parameters creates a perfect match between theory and simulations.

In chapter 5, we studied the improvements brought by pipelining the data I/O operations with the processing operations, both for tertiary storage I/O and for disk I/O. We showed that the improvement is of the order of 30% for the job splitting and for the out of order scheduling policies. However, the speedup improvement is smaller for the delayed scheduling policy due to the long delays it suffers from. These results will be published in a special issue of the International Journal of Computational Science and Engineering in 2006 [PHed].

In chapter 6, we describe how the proposed algorithms can be implemented in the context of the LHCb experiment at CERN, using the parallelization tools developed at the EPFL. We show that the architecture of both the LHCb framework and the DPS parallelization framework allow a very easy introduction of parallelism into the existing system. A first prototype was implemented and all scheduling policies can be easily plugged into it.

The next step is to port this prototype to recent versions of the LHCb framework and check our results against real computations. However, this requires studying how far the results presented in the present thesis at a reduced scale also apply to the full blown system. Since each simulation presented here is based on several years of simulated activity.

Appendix A

Computation of $A(1)$ and $\left.\frac{\partial A(z)}{z}\right|_{z=1}$

These values are used on page 65 and 66 and we give here the details of their computations. The definition of $A(z)$ is

$$A(z) = \frac{z^{mr} - 1}{z - 1} \tag{A.1}$$

We compute $A(1)$, according to the l'Hospital rule :

$$\begin{aligned} A(z)|_{z=1} &= \left.\frac{z^{mr} - 1}{z - 1}\right|_{z=1} \\ &= \frac{\left.\frac{\partial(z^{mr}-1)}{\partial z}\right|_{z=1}}{\left.\frac{\partial(z-1)}{\partial z}\right|_{z=1}} \\ &= \frac{mrz^{mr-1}|_{z=1}}{1} \\ A(1) &= mr \end{aligned} \tag{A.2}$$

The same technique can be applied to the computation of $\left.\frac{\partial A(z)}{z}\right|_{z=1}$:

$$\begin{aligned}
\left.\frac{\partial A(z)}{z}\right|_{z=1} &= \left.\frac{\partial \frac{z^{mr}-1}{z-1}}{\partial z}\right|_{z=1} \\
&= \left.\frac{mrz^{mr-1}(z-1)-(z^{mr}-1)}{(z-1)^2}\right|_{z=1} \\
&= \left.\frac{\frac{\partial\left[mrz^{mr-1}(z-1)-(z^{mr}-1)\right]}{\partial z}}{\frac{\partial[(z-1)^2]}{\partial z}}\right|_{z=1} \\
&= \left.\frac{mr(mr-1)z^{mr-2}(z-1)+mrz^{mr-1}-mrz^{mr-1}}{2(z-1)}\right|_{z=1} \\
&= \left.\frac{mr(mr-1)z^{mr-2}}{2}\right|_{z=1} \\
\left.\frac{\partial A(z)}{z}\right|_{z=1} &= \frac{mr(mr-1)}{2} \qquad \text{(A.3)}
\end{aligned}$$

Appendix B

Data utilisation distribution

Let us compute the probability p of having two jobs with overlapping data first in the case of a uniform data utilisation distribution and then in the case of the data utilisation distribution of Figure 3.2 on page 42.

B.1 Uniform data utilisation distribution

Two jobs have data in common if and only if their data segments overlap. We will first derive the probability p_f for 2 segments to overlap in the case of a uniform data utilisation distribution.

Let L be the average length of a job data segment and D the total length of the data space. We consider a given segment in the dataspace of length l starting at point a. Overlapping segments among the ones of length λ are the ones with a starting point between $a - \lambda$ and $a + l$ (See Figure B.1).

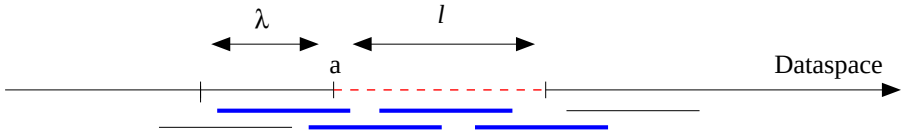

Figure B.1: Probability of having 2 overlapping segments. Segments having an overlap with the dashed one are thick.

Let f be the distribution of segment lengths. If we neglect border effects, the probability of having overlapping segments is :

$$p_f = \int_{l=0}^{\infty} \int_{\lambda=0}^{\infty} \frac{l+\lambda}{D} f(l)\ f(\lambda)\ dl\ d\lambda \tag{B.1}$$

By definition of the segment length distribution f and of the mean segment length L, we have :

$$\int_{l=0}^{\infty} f(l)\, dl = 1 \tag{B.2}$$

$$\int_{l=0}^{\infty} l\, f(l) dl = L \tag{B.3}$$

leading to :

$$\begin{aligned} p_f &= \int_{\lambda=0}^{\infty} \frac{\int_{l=0}^{\infty} l\, f(l)\, dl + \lambda \int_{l=0}^{\infty} f(l)\, dl}{D} f(\lambda) d\lambda \\ &= \int_{\lambda=0}^{D} \frac{L+\lambda}{D} f(\lambda)\, d\lambda \\ &= \frac{L \int_{\lambda=0}^{D} f(\lambda)\, d\lambda + \int_{\lambda=0}^{D} \lambda\, f(\lambda)\, d\lambda}{D} \end{aligned} \tag{B.4}$$

And with overlapping segments of average length L, we obtain :

$$p_f = \frac{2\, L}{D} \tag{B.5}$$

With a data space size $D = 2\, TB$ and an average segment length $L = 24\, GB$, we obtain an overlap probability $p_f = 2.4\%$.

B.2 Realistic data utilisation distribution

In the case of the data segment utilisation distribution of Figure B.2, the computation is approximately the same as for a uniform distribution except that we have to distinguish different zones depending on the data utilisation value. For each zone, the result of the uniform distribution case can be applied.

Figure B.2 names the different zones. We call D_X the length of zone X and ρ_X the probability that a job accesses a data segment belonging to zone X (we neglect here the overlapping regions). Values for ρ_X and D_X are given in Table B.1.

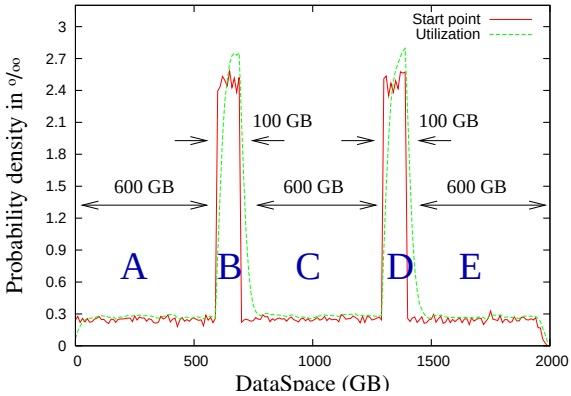

Figure B.2: Start point and data utilisation distribution

$\rho_A = \frac{1}{6}$	$D_A = D * 30\%$
$\rho_B = \frac{1}{4}$	$D_B = D * 5\%$
$\rho_C = \frac{1}{6}$	$D_C = D * 30\%$
$\rho_D = \frac{1}{4}$	$D_D = D * 5\%$
$\rho_E = \frac{1}{6}$	$D_E = D * 30\%$

Table B.1: Probability that a job belongs to a given zone and length of the zone. The probability of accessing a given data segment zone is given by the utilisation density of Figure B.2 multiplied by the data range size.

We can then derive p :

$$p = \sum_{X \in \{A,B,C,D,E\}} \rho_X^2 \frac{2\,L}{D_X} \tag{B.6}$$

$$= 3\left(\frac{1}{6}\right)^2 \frac{2\,L}{.3\,D} + 2\left(\frac{1}{4}\right)^2 \frac{2\,L}{.05\,D}$$

$$\approx 2.78\,\frac{2\,L}{D} \tag{B.7}$$

Providing that $D = 2TB$ and $L = 24GB$, we get $p = 6.67\%$.

List of Figures

List of Tables

Bibliography

[AGR03] Micah Adler, Ying Gong, and Arnold L. Rosenberg. Optimal sharing of bags of tasks in heterogeneous clusters. In *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, Scheduling I, pages 1–10. ACM Press, 2003.

[APF+03] Antonio Amorim, Luis Pedro, Han Fei, Nuno Almeida, Paulo Trezentos, and Jaime E. Villate. Grid-brick event processing framework in geps. In *Computing for High Energy Physics Conference 2003*, March 2003. published in eConf C0303241:THAT004.

[AS97] Stergios V. Anastasiadis and Kenneth C. Sevcik. Parallel application scheduling on networks of workstations. *Journal of Parallel and Distributed Computing*, 43(2):109–124, 1997.

[BBS+94] Robert Bennett, Kelvin Bryant, Alan Sussman, Raja Das, and Joel Saltz. Jovian: A framework for optimizing parallel I/O. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 10–20, Mississippi State, MS, 1994. IEEE Computer Society Press.

[BFR92] Yair Bartal, Amos Fiat, and Yuval Rabani. Competitive algorithms for distributed data management. In *24th Annual ACM STOC*, pages 39–50, 1992.

[BGR03] Veeravalli Bharadwaj, Debasish Ghose, and Thomas G. Robertazzi. Divisible load theory: A new paradigm for load scheduling in distributed systems. *Cluster Computing, Special Issue on Divisible Load Scheduling in Cluster Computing*, 6(1):7–17, January 2003. Kluwer Academic Publishers.

[BKK96] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. The X-tree: an index structure for high-dimensional data. In *VLDB'96, Proc.*

of 22th International Conference on Very Large Data Bases, pages 28–39, 1996.

[CER05] CERN web pages. The Castor project. http://castor.cern.ch, 2005.

[cg05] CERN LHCb computing group. Gaudi Framework. http://cern.ch/Gaudi, 2005.

[Col00a] LHCb Collaboration. Calorimeters TDR. Technical report, CERN, September 2000. http://lhcb.web.cern.ch/lhcb/TDR/TDR.htm.

[Col00b] LHCb Collaboration. Rich TDR. Technical report, CERN, September 2000. http://lhcb.web.cern.ch/lhcb/TDR/TDR.htm.

[Col01a] LHCb Collaboration. Outer Tracker TDR. Technical report, CERN, September 2001. http://lhcb.web.cern.ch/lhcb/TDR/TDR.htm.

[Col01b] LHCb Collaboration. VELO TDR. Technical report, CERN, May 2001. http://lhcb.web.cern.ch/lhcb/TDR/TDR.htm.

[Col02] LHCb Collaboration. Outer Tracker TDR. Technical report, CERN, November 2002. http://lhcb.web.cern.ch/lhcb/TDR/TDR.htm.

[Dat05] The datagrid project. http://eu-datagrid.web.cern.ch/eu-datagrid/, 2005.

[EGE05] Enabling grid for e-science. http://egee-intranet.web.cern.ch/egee-intranet/gateway.html/, 2005.

[Eur05] The eurogrid project. http://www.eurogrid.org//, 2005.

[FK97] Ian Foster and Carl Kesselman. Globus: A Metacomputing Infrastructure Toolkit. In *Intl J. Supercomputer Applications*, volume 11(2), pages 115–128, 1997.

[FS00] Rudolf Fleischer and Steven S. Seiden. New results for online page replication. In *Proceedings of the International Workshop on Approximation Algorithms for Combinatorial Optimization (APPROX '00)*, pages 144–154. Springer Lecture Notes in Computer Science, 2000.

[Gal99] Thomas Galla. *Cluster Simulation in Time-Triggered Real-Time Systems*. PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, 1999.

[GH03] S. Gerlach and R.D. Hersch. DPS - Dynamic Parallel Schedules. In *Proc. of 8th Int'l Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2003), 17th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 15–24. IEEE Press, 2003.

[Gut84] Antonin Guttman. R-trees : A dynamic indexing structure for spatial searching. In *Proceedings of the ACM SIGMOD international conference on Management of data*, pages 47–57, 1984.

[KKKC02] I. Kadayif, M. Kandemir, I. Kolcu, and G. Chen. Locality-conscious process scheduling in embedded systems. In *Proceedings of the tenth international symposium on Hardware/software codesign*, System design methods: scheduling advances, pages 193–198. ACM Press, 2002.

[Kle76] L. Kleinrock. *Queueing Systems*. Wiley-Interscience, 1976.

[KN93] J. Kaplan and M. L. Nelson. A comparison of queueing, cluster, and distributed computing systems. NASA Technical Memorandum 109025, NASA LaRC, october 1993. http://citeseer.nj.nec.com/-kaplan94comparison.html.

[KS97] Norio Katayama and Shin'ichi Satoh. The SR-tree: an index structure for high-dimensional nearest neighbor queries. In *Proceedings of the ACM SIGMOD international conference on Management of data*, pages 369–380, May 1997.

[ML96] Jussi Myllymaki and Miron Livny. Efficient Buffering for Concurrent Disk and Tape I/O. *Performance Evaluation*, 27/28(4):453–471, 1996.

[NB90] B. Seeger N. Beckmann, H-P. Kriegel R. Schneider. The R*-tree : An Efficient and Robust Access Method for Points and Rectangles. In *Proceedings of the ACM SIGMOD international conference on Management of data*, pages 322–331, 1990.

[NEO03] Harvey Newman, Mark H. Ellisman, and John A. Orcutt. Data-intensive e-science frontier research. *Commun. ACM*, 46(11):68–77, 2003.

[PH04] Sébastien Ponce and Roger D. Hersch. Parallelization and scheduling of data intensive particle physics analysis jobs on clusters of pcs. In *18th*

International Parallel and Distributed Processing Symposium (IPDPS 2004), CD-ROM / Abstracts Proceedings, 26-30 April 2004, Santa Fe, New Mexico, USA, volume 0-7695-2132-0, page 233. IEEE Computer Society, 2004. http://dblp.uni-trier.de.

[PHed] Sébastien Ponce and Roger D. Hersch. Scheduling data intensive particle physics analysis jobs on clusters of pcs. *International Journal of Computational Science and Engineering*, 2006, to be published.

[PVH02] Sébastien Ponce, Pere Mato Vila, and Roger D. Hersch. Indexing and selection of data items in huge data sets by constructing and accessing tag collections. In *Proceedings of nineteenth IEEE Symposium on Mass Storage Systems and tenth Goddard Conference on Mass Storage Systems and Technologies*, pages 181–192, April 2002.

[Sin05] Simon Singh. *Big Bang : The Origin of the universe*. Fourth Estate, 2005. ISBN: 0007162200.

[tea06] ROOT team. ROOT web page. http://root.cern.ch, 2006.

[TF98] Peter Triantafillou and Christos Faloutsos. Overlay striping and optimal parallel I/O for modern applications. *Parallel Computing*, 24(1):21–43, 1998.

[UKSS98] Mustafa Uysal, Tahsin M. Kurc, Alan Sussman, and Joel H. Saltz. A performance prediction framework for data intensive applications on large scale parallel machines. In *Proc. of 4th Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR98)*, volume 1511 of *Lecture Notes in Computer Science*, pages 243–258. Springer-Verlag, Pittsburgh, PA, 1998.

[Vic87] Robert Vich. *Z Transform Theory and Applications*. Springer, June 1987. ISBN: 9027719179.

[WB97] Roger Weber and Stephen Blott. An approximation-based data structure for similarity search. Technical Report 24, ESPRIT project HERMES (no. 9141), October 1997. Available at http://www-dbs.ethz.ch/∼weber/paper/HTR24.ps.

[wp05a] CERN web pages. A matter of symmetry. http://lhcb-public.web.cern.ch/lhcb-public/html/symmetry.htm, 2005.

[wp05b] CERN web pages. CERN Web portal. http://www.cern.ch/, 2005.

[wp05c] CERN web pages. What is CP-violation? http://lhcb-public.web.cern.ch/lhcb-public/html/introduction.htm, 2005.

[wp05d] LHCb web pages. Experiments in B-physics. http://lhcb-public.web-.cern.ch/lhcb-public/html/bphysicsexpts.htm, 2005.

[WSB98] Roger Weber, Hans-Jörg Schek, and Stephen Blott. A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces. In Ashish Gupta, Oded Shmueli, and Jennifer Widom, editors, *VLDB'98, Proc. of 24th International Conference on Very Large Data Bases*, pages 194–205. Morgan Kaufmann, 1998.

[ZBWS98] B. B. Zhou, R. P. Brent, D. Walsh, and K. Suzaki. Job scheduling strategies for networks of workstations. *Lecture Notes in Computer Science*, LNCS 1459:143–157, 1998.

Biography

Sébastien Ponce was born on December 21^{st}, 1975 in Chambéry, France. From September 1995 to June 1998, he studied at Ecole Polytechnique, in Palaiseau, France where he obtained his first engineering diploma. He did his diploma work at NUM, a subsidiary company of Schneider Electric located near Paris, France. He worked on the optimization of numeric commands for machine tools. From September 1998 to June 2000 he studied at Ecole Nationale Supérieure des Télécommunication (ENST) in Paris, France, where he obtained a second engineering diploma. He did a first diploma work during the year at ENST, developing a complete, open source CORBA Object Request Broker in Ada, now integrated into polyORB tool available as part of the GNAT products. He then did a second diploma work at Sun Microsystems in Palo Alto, California during 6 months, working on the integration of Netscape products into the Sun Microsystems product line.

In September 2000, Sébastien came to CERN, Geneva as a member of the LHCb computing team. He worked on the geometrical description of the LHCb detector based on XML and its automatic conversion into C++ objects. At the end of 2001, he started a phd work in collaboration with the Peripheral Systems Laboratory of EPFL concerning the parallelization of the LHCb framework (Gaudi) on large clusters of PCs. This thesis is the final result of this work, that was continued on his spare time after he left LHCb. In spring 2002, Sébastien became a CERN fellow, working on the maintenance of the Gaudi framework with a special expertise in compilers. Sébastien moved to the Information Technology division of CERN in September 2003 where he became member of the CASTOR developement team. CASTOR deals with automatic handling of tertiary storage, i.e. magnetic tapes for the CERN data (several petaBytes per year). Sébastien started the rewriting of some essential component of the CASTOR software that could not scale to the next generation of softwares and the Grid context. The "CASTOR 2" first version was tested at the beginning of 2005 and is now the base of the

CERN tertiary storage infrastructure. In April 2006, Sébastien became the leader of the CASTOR project. He is still leading the team dealing with the development and support of the CASTOR 2 product.

Publications

[1] Sébastien Ponce and Roger D. Hersch. Scheduling data intensive particle physics analysis jobs on clusters of pcs. *International Journal of Computational Science and Engineering*, 2006, to be published.

[2] Olof Bärring, Benjamin Couturier, Jean-Damien Durand, Emil Knezo, and Sébastien Ponce. Storage resource sharing with castor. In *12th NASA Goddard/21st IEEE Conference on Mass Storage Systems and Technologies, Proceedings*. IEEE Computer society, April 2004. http://castor.web.cern.ch/castor/PRESENTATIONS/2004.

[3] Olof Bärring, Benjamin Couturier, Jean-Damien Durand, Emil Knezo, Sébastien Ponce, and Tim Smith. Castor: Operational issues and new developments. In *Computing for High Energy Physics Conference, Proceedings*, September 2004. http://castor.web.cern.ch/castor/PRESENTATIONS/2004.

[4] Sébastien Ponce and Roger D. Hersch. Parallelization and scheduling of data intensive particle physics analysis jobs on clusters of pcs. In *18th International Parallel and Distributed Processing Symposium (IPDPS 2004), CD-ROM / Abstracts Proceedings, 26-30 April 2004, Santa Fe, New Mexico, USA*, volume 0-7695-2132-0, page 233. IEEE Computer Society, 2004. http://dblp.uni-trier.de.

[5] Andrei Tsaregorodtsev, Markus Frank, Joel Closier, Sébastien Ponce, and Eric van Herwijnen. Dirac : Distributed implementation with remote agent control. In *Computing for High Energy Physics Conference 2003*, March 2003. published in eConf C0303241:TUAT006.

[6] Sébastien Ponce, Ivan Belyaev, Pere Mato Vila, and Andrea Valassi. Detector description framework in lhcb. In *Computing for High Energy Physics Conference 2003*, March 2003. published in eConf C0303241:THJT007.

[7] Sébastien Ponce, Pere Mato Vila, and Roger D. Hersch. Indexing and selection of data items in huge data sets by constructing and accessing tag collections. In *Proceedings of nineteenth IEEE Symposium on Mass Storage*

Systems and tenth Goddard Conference on Mass Storage Systems and Technologies, pages 181–192, April 2002.

[8] Marco Cattaneo, Markus Frank, Pere Mato, Sébastien Ponce, Florence Ranjard, Ivan Belyaev, Christian Arnault, Paolo Calafiura, Christopher Day, Charles Leggett, Massimo Marino, David Quarrie, and Craig Tull. Status of the gaudi event-processing framework. In Science Press, editor, *Computing for High Energy Physics Conference 2001*, pages 209–212, September 2001.

[9] Fabien Azavant, Jean-Marie Cottin, Vincent Niebel, Laurent Pautet, Sébastien Ponce, Thomas Quinot, and Samuel Tardieu. CORBA and CORBA Services for DSA. In *ACM SigAda'99*, October 1999.

Other References

[1] The AdaBroker project. http://www.adapower.net/adabroker/.

[2] The PolyORB project. http://polyorb.objectweb.org/.

[3] AdaCore, The GNAT Pro Company . http://www.gnat.com/home/.

[4] The LHCb Web pages. http://lhcb.cern.ch/.

[5] Gaudi Framework. http://cern.ch/Gaudi.

[6] CERN web pages. The Castor project. http://castor.cern.ch, 2005.